UiO : Faculty of Mathematics and Natural Sciences
University of Oslo

Parallelisation of Hierarchical Clustering Algorithms for Metagenomics

Mimi Tantono

Master's Thesis, June 2015

Parallelisation of Hierarchical Clustering Algorithms for Metagenomics

Mimi Tantono

June 29, 2015

Preface

I would like to express the deepest gratitude to my thesis supervisor, Torbjørn Rognes, for his continuous support and encouragement during the work on this thesis. Apart from that, I would also like to acknowledge the time and effort that he has spent for implementing the sub-sampling feature in vsearch that has accommodated the experiments on partial datasets in this thesis. And most of all, without his guidance and valuable advice, the completion of this thesis would not have been possible.

In addition, I would also like to thank my close friends and family for all of their love and understanding that has supported me through the entire process.

Abstract

Metagenomics is the investigation of genetic samples directly obtained from the environment. Driven by the rapid development of DNA sequencing technology and continuous reductions in sequencing costs, studies in metagenomics become popular over the past few years with the potential to discover novel knowledge in many fields through analysing the diversity of microbial ecology.

The availability of large-scale datasets increases the challenge in data analysis, especially for hierarchical clustering that has a quadratic time complexity. This thesis presents the design and implementation of a parallelisation method for single-linkage hierarchical clustering for metagenomics data. Using 16 parallel threads, p-swarm was measured to achieve 11 times of speedup. This result shows a significant improvement of execution time while preserving the quality of exact and unsupervised clustering, which makes it possible to hierarchically cluster a larger dataset, for example TARA dataset which consists of nearly 10 million amplicons in just a few hours. Moreover, our method may be extended to a distributed computing model that could further increase the scalability and the capacity to cluster a larger volume of dataset.

Availability: P-swarm was implemented in C++ and source code is available on https://github.com/mimitantono/p-swarm under the GNU Affero GPL.

Keywords: hierarchical clustering; single-linkage; parallel programming; metagenomics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The approximated amount of prokaryotes and their cellular carbon on earth is $4 - 6 \times 10^{30}$ cells, most occurs in the open ocean (1.2×10^{29}), in soil (2.6×10^{29}), in oceanic (3.5×10^{30}), and terrestrial subsurfaces ($0.25 - 2.5 \times 10^{30}$) [Whitman, Coleman, & Wiebe, 1998]. Although the number of prokaryotes on animals (including human) was found not to constitute a great proportion of prokaryotes, they play an important role in nutrition and disease. For instance the human body is composed of approximately 10^{13} eukaryotic cells, while the body surfaces and gastrointestinal canals, i.e. oesophagus, stomach, small intestines, cecum, and large intestines - of humans may be colonised by as many as 10^{14} indigenous prokaryotic and eukaryotic microbial cells [Savage, 1977]. These residents microbiome and human genome somehow form a mutualistic symbiosis relationship to a certain extent that they are dependent with each other where the disruption of one may affect the well-being of the others, for example, microbiome provide enzymes for digestion, and overgrowth of intestinal flora may cause irritable bowel syndrome [Nelson, 2011].

Over more than 99% of microorganisms within all microbial groups (i.e. bacteria, archaea, fungi, viruses, algae and protozoa) are "unculturable" with the current laboratory techniques. For the ones that can be grown in the lab's artificial environments, the result would unlikely reflect its original gene expression, protein and metabolite profiles in nature. Therefore, the approach to look at microorganisms at their native habitats is crucial for understanding their functions and characteristics. With the development of high-throughput sequencing (HTS) technologies, an enormous number of sequences data is now available without having to go through the process of cultivation. It enables the study of metagenomics, it opens many potential discoveries related to natural microbial diversity and its correlation with stability in natural environments, and it also shifts the weight of analysis to the area of data analysis.

One of the main steps in analysing these redundant biological data

is to cluster similar OTUs (Operational Taxonomy Units) that are close to each other but far from the others, so that further expensive analyses can be performed on the set of grouped data. Hierarchical clustering offers an unsupervised and non-parametric approach to this step but is subject to more challenges due to its complexity in time and space in addition to the high-cost pairwise alignment. To anticipate the increasing number of sequences that are now available to be processed, there is a need to work out an approach to attaining scalability with parallelisation while preserving the clustering accuracy of the hierarchical clustering algorithm.

1.2 Problem Statement

Based on the motivations mentioned above, the objective of this study is as following.

- To review the strategies for parallelisation of single-linkage clustering in the context of metagenomics.
- To design and implement such an efficient strategy.
- To evaluate and compare it to existing tools.

Chapter 2

Background

2.1 Metagenomics

Metagenomics was first introduced when in 1998 Handelsmann et al. explored the concept of studying soil microbes by directly accessing its genomes and bypassing laboratories culture [Gilbert & Dupont, 2011]. Since the samples are studied directly without having to rely on microbiological culture, it encourages the studies of organisms that are not easily cultured in the laboratory in addition to saving time and cost for collecting samples. As mentioned before, 99% of all microorganisms are not culturable while according to Eckburg et al. only 20% of human intestinal microbiota are culturable (as cited in [Nelson, 2011])

A pilot project of environmental study was conducted in 2004 by J. Craig Venter Institute in the Sargasso Sea – named before the *Sargassum* brown seaweed genus that is floating on the surface – which is located in Atlantic Ocean near Bermuda [Sleator, Shortall, & Hill, 2008]. Using whole-genome shotgun sequencing, the outcome of this "largest-ever" metagenomics project at the time was the identification of “1,800 new species and more than 1.2 million new genes” [Larkman, 2007]. Apart from its application on the natural environment, metagenomics can also be performed on several locations of the human body, e.g. skin, mouth, intestine, with the goal to potentially aid improving human health.

2.2 Studies Related to Metagenomics

Microbial ecology focuses on two areas of study: (i) microbial diversity, (e.g. phylogenetic diversity, species diversity, genotype diversity, gene diversity, evolutional diversity, metabolic diversity, functional diversity) and (ii) microbial activity [Xu, 2006]. The characteristics of metagenomics that does not require DNA culture has driven more studies about microbial diversity that are rich with promises in bringing useful applications on the environment and human life. There are two initiatives related to metagenomics study which had eventually expanded to worldwide where researchers from all around the world

collaborate by making their data and finding available to be freely used.

2.2.1 The Human Microbiome Project (HMP)

In the long run the study of human microbiome is expected to provide potential interventions on altering the state of one's microbiome compositions using antibiotics, prebiotics (short-chain carbohydrates), probiotics (microorganisms such as bacteria or yeast), or synbiotics (combinations of the former three components) to enhance human health or even in curing and preventing diseases [Nelson, 2011]. Furthermore, the evidence indicates that either pathogen or beneficial microorganisms evolve as a human does, hence to understand human microbiome is important and it will give much interest into the medicine related fields of study.

With the reasons mentioned above, the interest in studying the microbiome increased with the advent of microbial communities metagenomics analyses and the ability to produce the whole genome sequence of bacterias. In 2008 this became a reality when HMP was selected as an initiative in the National Institute of Health (NIH) Roadmap for Biomedical Research with a total investment over $150 million with the aims to produce reference genome sequences for microbes, determine their structures in the 18 body sites of healthy subjects, develop new laboratory and computational approaches, perform projects investigating human diseases, and investigate the ethical, legal and social impact of these research [Weinstock, 2011].

2.2.2 The Earth Microbiome Project (EMP)

The concept of EMP was born as the main outcome of The Terabase Metagenomics Workshop in summer 2010, and its name was given to show respect to HMP as the pioneer in the field. The workshop which was sponsored by the Institute for Computing in Science (ICiS) aimed to address the challenge of finding the best possible way to utilise the discovery of next-generation sequencing platforms such as the Illumina which can sequence 250 billion DNA base pairs in as short as 8 days [Gilbert et al., 2010]. Further analysis such as metagenomics, meta-transcriptomics and amplicon sequencing will be executed on the collected dataset to assemble a "global Gene Atlas" which consists of the information of proteins, environmental metabolic model, and a portal for data visualisation.

Following the workshop, the EMP was launched consequently in August 2010 with the objectives mentioned above. Through "crowdsourcing, soliciting donations of samples" from more than 200 researchers around the world within numerous different microbial ecology disciplines – human animal, plan, terrestrial, marine, freshwater, sediment, air, built-environment, and every intersection of these ecosystems – in approximately 4 years (as of July 2014) EMP had acquired and processed over 30,000 environmental samples consist of more than 40 dif-

ferent ecologies of the earth, generated 16S rRNA amplicon data and releasing them to QIIME (Quantitative Insights into Microbial Ecology) database [Gilbert, Jansson, & Knight, 2014]. EMP's mission is to continue to grow and adapt, exploring metagenomics analysis, and adding new avenues including potentially extra-terrestrial locations.

2.2.3 Example of Applications of Metagenomics on Microbial Ecology

We collected some examples of potential uses of metagenomics on microbial ecology as following:

2.2.3.1 Application on Plant-Microbe Interactions

Soils are the most genetically rich environments on earth and because of that its metagenomics analysis was fairly neglected due to the technical difficulties of DNA analyses. Plant and soil microbes have a strong relationship with each other, therefore studying soil microbes will lead to the improvement of planting and cultivation techniques, immunisation and fertilisation resulting in healthy and productive crops (as cited in [Kuiper, Lagendijk, Bloemberg, & Lugtenberg, 2004]).

2.2.3.2 Application on Bioremediation

The concept of bioremediation is to clean up the polluted environment with as little human intervention as possible so that the advantages of cost saving and permanent effect can be achieved [George, Stenuit, Agathos, & Marco, 2010]. Some bacteria and funguses have the metabolic characteristics which chemically purify *hazardous pollutants* and metagenomics is a convenient tool to understand these physicochemical characteristics.

2.2.3.3 Application on Industrial Bioproducts

Microbial genomics were already used before to develop enzymes that are useful in biotechnology such as: antibiotics and secondary metabolites, organonitriles and industrial synthesis (synthesis of plastics, fibre, fumigant, dyestuffs, etc.), food processing (e.g. modification of lipids in the dairy products manufacturing), and many more [Wong, 2010]. There are still a large number of unknown enzymes that yet to be discovered and will contribute to boosting the efficiency of industrial processes.

2.3 Sequencing of DNA

DNA sequencing is the process of identifying the occurrences of four different types of nucleotides namely *adenine, guanine, cytosine*, and *thymine* within a DNA strand of the cell of an organism.

2.3.1 History of Influential Sequencing Technology

In the early days, the first sequencing technology was developed by Frederick Sanger in 1977, referred as "Sanger" sequencing or Chain-terminator sequencing. This sequencing technology was considered to be easy to use and reliable to determine nucleotide sequences in a single stranded DNA, therefore it was widely used for around 25 years. However, Sanger technology is an expensive and low-throughput technology aside from the limitation of it being biologically biased [Bragg & Tyson, 2014]. In order for the method to give a reliable result, one must satisfy various criteria related to compatibility of the samples to be sequenced with *Escherichia Coli* as the DNA template. Despite these reasons, this technique is still useful on smaller-scale projects that require long contiguous DNA sequence reads.

Pyrosequencing was invented 20 years later, and precisely in 2005 Roche 454 successfully developed a commercial DNA sequencer with the performance to generate about 500 million bases of raw sequence in just a few hours (as cited in [Pettersson, Lundeberg, & Ahmadian, 2009]). A few years later during 2008, Illumina and Applied Biosystems SOLiD (Sequencing by Oligo Ligation and Detection) have introduced the sequencing systems that offer even higher throughput at billions of bases in a single run. To obtain such the capability of generating such a generous number of base pairs, both of the novel methods rely on the parallelisation of spatially separated clonal amplicons that leads into a much higher throughput.

The comparison of characteristics and details of each sequencing technology are shown on table 2.1 [Glenn, 2011; Liu et al., 2012]. The cost, accuracy, read length, runtime and output of each technology were retrieved from "2014 NGS Field Guide: Overview"[1]. The numbers represented in the table were taken from the following instruments that were selected as the example from each platform:

- 454: 454 FLX+
- Illumina: Illumina HiSeq 2500 - high output v4
- IonTorrent: Ion Torrent - PGM 318 chip

Sanger and PacBio each has only one model: Applied Biosystems 3730 (capillary) and Pacific Biosciences RS II.

[1] http://www.molecularecologist.com/next-gen-fieldguide-2014/

Platform	Sanger	454	Illumina	Ion Torrent	PacBio
Current Company	Applied Biosystems	Roche	Illumina	Life Technologies	Pacific Bioscience
Former Company	Life Technology	454	Solexa	Ion Torrent	N/A
Sequencing Method	Chain-termination	Synthesis (Pyrosequencing)	Synthesis	Synthesis (H+ detection)	Synthesis
Claim to Fame	High quality; Long read length	First NGS; Long reads	First short-read sequencer; Current leader in advantages	First Post-light sequencer; First system < $100.000	First real-time single-molecule sequencing
Cost / million bp	$2,307	$9.5	$0.06, $0.03	$0.79, $0.46	$1.11
Final error rate	0.1-1%	1%	~ 0.1%	~ 1%	≤ 1%
Read Length	650 bp	650bp	50 bp / 250 bp	200 bp / 400 bp	3,000 bp
Time / Run	2 hours	20 hours	40 hours / 6 days	4.4 hours / 7.3 hours	2 hours
Output / Run	62.4 Kbp	650 Mbp	100 Gbp / 500 Gbp	950 Mbp / 1.9 Gbp	90 Mbp

Table 2.1: Comparison of Sequencing Instruments

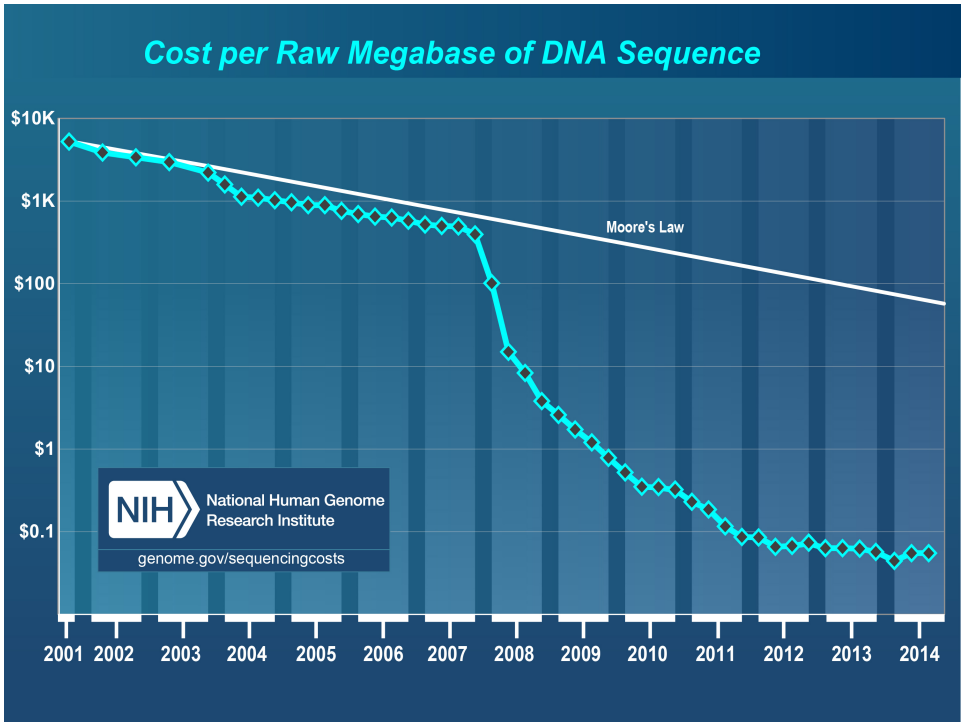


Figure 2.1: Cost of DNA Sequencing from 2001-2014
Source: National Human Genome Research Institute. [2015, June]. Retrieved from http://www.genome.gov/images/content/cost_megabase_.jpg

The discovery of HTS (High-Throughput Sequencing) has opened the window of more scientific research, expanding our knowledge into the area that was not available before due to the limitation of sequencing technology. The progressive development of sequencing methods not only allowed researchers to directly sequence without having to culture the environmental or human body samples, it also substantially reduces the cost of DNA sequencing. Figure 2.1 shows that the cost of sequencing in the past 10 years has dropped more than predicted by Moore's law (computing power would double every two years).

2.3.2 FASTA Sequence File Format

There are many file formats exists in the field of Bioinformatics. One of them that is the oldest and simplest is FASTA sequence format [BioPerl, 2014], which is a text file containing one or more DNA or protein sequences. Each sequence is represented by two lines; the first line as the header must be marked by character > followed by a description, and the second line contains symbols of amino acids or nucleotides, usually in capital.

There is no formal standard of what a header line should contain, and which character to be used as the separator. Most projects usually have their own way to specify some information in the header line. For example an "NCBI" (National Center for Biotechnology Information) formatted FASTA sequence header usually includes the

database identifier number, database code, accession number (identifier number for a sequence), and Locus name, delimited by a pipe character, e.g.:

```
>gi|142864|gb|M10040.1|BACDNAE B.subtilis dnaE gene
    encoding DNA primase, complete cds
```

Most of the heuristic clustering tools such as CD-HIT, USEARCH, VSEARCH, DNACLUST, and so on, recognise abundance by the label "size", which is placed after the description of the sequence and delimited by a semicolon character, e.g.:

```
>0b65e14109c71683d27299a3fc5fe362fca9f18d;size=2034;
```

There is also no formal standard file extensions for a FASTA sequence file. Usually `*.fas` or `*.fasta` is used for any generic fasta file, `*.fna` for any fasta containing nucleic acids, `*.ffn` for coding regions of a genome, `*.faa` for amino acids, and `*.frn` for non-coding RNA regions.

2.4 Clustering on Biological Data

Given a set S of n vertices in $\mathbb{R}^d$, a clustering problem is defined as the operation of classifying S into k clusters – usually unsupervised – such that the points in each cluster are either close to each other within the distance of d or close to some *cluster center* or *centroid* within the distance of $d/2$. In the context of data-mining where there exists an abundant number of raw data, clustering is regularly used for compression, reducing redundancy, and grouping of data so that a more expensive analysis can later be performed on the smaller amount of data.

Unsupervised clustering is one of the approaches that are used in metagenomics to assign DNA sequences into operational taxonomy units (OTUs) or phylotypes — a unit of organism in microbial diversity — in order to estimate the richness and diversity of a microbiome community. OTUs in metagenomics are commonly specified based on the genetic distances between sequences since DNA sequences are the only available data for these organisms.

Clustering is relatively more complex when performed on biological data because a sequence database is a high dimensional data that exists in non-metric spaces. Given sequences x, y, and z and a scoring function d to measure the similarity between one sequence with another, the triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$ does not strictly hold [Cai & Sun, 2011]. However, a Monte Carlo experiment was performed by Cai et al. to measure the significance of the violation and it was found out that there were only 7 out of 100K trials that did not comply with the inequality.

On the other hand, HTS uses PCR (Polymerase Chain Reaction) to amplify DNA samples, and this technique is prone to produce a chimeric sequence. As cited by Edgar et al., up to 46% of 16S database is composed of chimeric sequences, mainly chimeras

with two segments (bimeras) and sometimes chimeras with more than two segments (multimeras) [Edgar, Haas, Clemente, Quince, & Knight, 2011]. Existing studies indicate that without preprocessing, clustering may give an overestimation of OTUs while data corrected with preprocessing would significantly improve the quality of the clustering [May, Abeln, Crielaard, Heringa, & Brandt, 2014; Bonder, Abeln, Zaura, & Brandt, 2012]. This shows as well that articulation processes like preprocessing prior or postprocessing succeeding to the clustering itself must not be neglected. QIIME is a popular open-source bioinformatics pipeline on which the entire workflow of data analysis – from raw sequencing data to demultiplexing and quality filtering, OTU picking, taxonomic assignment, phylogenetic reconstruction, diversity analyses and visualisations – may be performed [Caporaso et al., 2010].

2.4.1 Clustering Algorithms

2.4.1.1 K-means Clustering

Clustering is a classic problem. In 1967 MacQueen presented a simple unsupervised algorithm called k-means clustering which can be extended to be applied on several problem domains beyond the Euclidean distance [MacQueen et al., 1967]. The idea (and also the drawback) of this algorithm is to assume that there exist k centroids – which could be defined randomly – and then iterate each vertex to measure the distance between itself and the available centroids. Later the vertex will be assigned to the nearest cluster – having the minimum distance with the centroid, and the centre of cluster shall be re-calculated accordingly since its members have changed [Polanski, 2007].

2.4.1.2 Hierarchical Clustering

Another solution to the clustering problem is the *hierarchical clustering* which constructs a minimum spanning tree called *dendrogram* by first calculating a symmetrical matrix containing the value of pairwise distances between each vertex with all of the other vertices. To obtain clusters, the tree is cut on some levels, i.e. on the edges with the maximum length decided based on some threshold of the clustering, and hence every cluster is, in fact, some subtree of the complete dendrogram [Polanski, 2007; Olson, 1995]. The maximum length which is also the distance between two clusters can be measured either in graph metrics, where the distances are defined by the cost function of the edges between vertices in the two clusters, or by geometric metrics where the distances are calculated between centroids. A graph metric is generally used in most of the existing clustering programs since it reflects well the characteristic of a biological dataset. The commonly used graph metrics are as following (illustrated on figure 2.2).

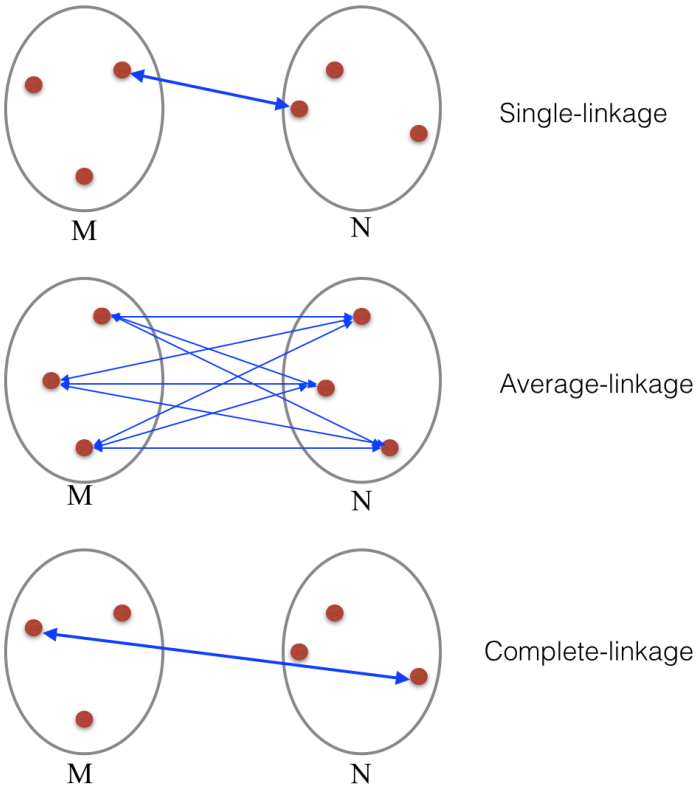


Figure 2.2: Graph Metrics in Hierarchical Clustering

- **single linkage:** minimum distance between vertices in the two clusters.

$$D(M,N) = \min_{m \in M, n \in N} d(m,n) \tag{2.1}$$

- **average linkage:** average distance between vertices in the two clusters.

$$D(M,N) = \frac{1}{|M||N|} \sum_{m \in M, n \in N} d(m,n) \tag{2.2}$$

- **complete linkage:** maximum distance between vertices in the two clusters.

$$D(M,N) = \max_{m \in M, n \in N} d(m,n) \tag{2.3}$$

There have been developed quite a handful number of existing traditional hierarchical clustering programs for biological data, using the most common strategies – grouping of 16S rRNA sequences into some *operational taxonomic unit* (OTU) – most of them having the complexity of at least $O(N^2)$ where N is the number of sequences, for example DOTUR in 2005 [Schloss & Handelsman, 2005] which was later integrated into Mothur in 2009 [Schloss et al., 2009], and ESPRIT in 2009 [Sun et al., 2009] succeeded by its improved version ESPRIT-tree [Cai & Sun, 2011], and SLP [Huse, Welch, Morrison, & Sogin,

2010]. ESPRIT was developed to reduce the complexity and memory requirement of Mothur, and ESPRIT-tree is an improved version of ESPRIT which achieves the same level of accuracy with its predecessor but having a quasi-linear computational complexity [Chen, Zhang, Cheng, Zhang, & Zhao, 2013] through employing a *k-mer filtering* scheme.

The time and space complexity which grows quadratically with the problem size becomes the main aspect that makes hierarchical clustering a challenging problem. Parallelising a hierarchical clustering algorithm is also a difficult problem as the general algorithm comprises a sequential workflow and every step requires access to the entire dataset. For this reason, heuristic clustering programs are currently more widely-used for clustering large amount of data.

2.4.1.3 Heuristic Clustering

A heuristic approach can be generally defined as the experience-based approach to problem solving through self-learning or trial and error so that it can produce results in a reasonable amount of time (in contrast to the classical approach). Usually, this method does not give an optimum solution, but just a local optimum or an estimation to the global optimum which is good enough for some applications.

Traditional heuristic clustering methods for biological data usually involve building a guide tree which is later used to construct the multiple sequence alignment (MSA) of every sequence [Ghodsi, Liu, & Pop, 2011]. Some of these methods select a sequence as a seed for the first cluster and then sequentially adding next sequences to this cluster if they are close, or form a new cluster if otherwise [Ghodsi et al., 2011; Chen et al., 2013].

There have been many heuristic clustering programs that were developed for biological data (in the order of the time published) such as: CD-HIT [Li & Godzik, 2006], UCLUST [Edgar, 2010], GramCluster [Russell, Way, Benson, & Sayood, 2010], DNACLUST [Ghodsi et al., 2011], CD-HIT (V4.6) [Fu, Niu, Zhu, Wu, & Li, 2012], sumaclust [Mercier, Boyer, Bonin, & Coissac, 2013] and VSEARCH [Flouri et al., 2015].

2.4.2 Algorithms and Methodologies of Several Heuristic Clustering Tools

All the heuristic tools CD-HIT, USEARCH, VSEARCH, DNACLUST, and sumaclust use a greedy incremental clustering algorithm as illustrated on figure 2.3. The clustering starts by assigning the first sequence in the input as the seed of the first cluster. Then, the seed will be compared with the remaining sequences. Every sequence that has a similarity above the required threshold will become a new group member of the cluster, otherwise it will be assigned as the seed of the next cluster. The same process happens again and again until every

sequence in the input belongs to a cluster. In this clustering method, the radius of the cluster is fixed to the identity threshold and the diameter of the cluster cannot be more than twice of the cluster's radius.

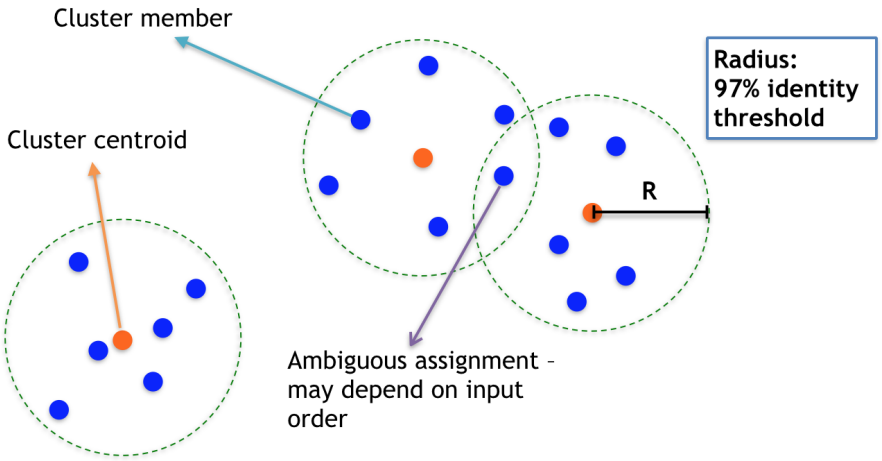


Figure 2.3: Centroid-based clustering based on identity threshold

Although all tools employ the same greedy clustering algorithm, there are small differences between each tool in terms of similarity definition, sorting of input, heuristic techniques to increase the speed or precision, and parallelisation mechanism. Almost all of the tools are highly customisable, and they are also equipped with a set of default options that become the outline of their special characteristics.

Each tool uses a similar heuristic method related to k-mer filtering to avoid expensive pairwise alignments. CD-HIT uses "short-word counting and indexing table" to rule out unnecessary alignments. In its shared-memory parallelised version written with OpenMP, one thread is assigned to handle the writing and reading to/from a global indexing table while the remaining threads are performing the clustering procedure [Fu et al., 2012]. USEARCH and VSEARCH use a similar filtering strategy by counting the number of common unique words or k-mers. *Sumaclust* is implemented with a "lossless k-mer filter" followed by a banded alignment algorithm that is very efficient for a high threshold, both filter and alignment are parallelised with SIMD [Mercier et al., 2013].

USEARCH compares the seed with the targets in a decreasing order by total common unique words between the seed and the target, the sorted list is referred as the "U vector". A search is terminated after some defined number of accepts (by default 1) and rejects (by default 8) because the total common unique words correlate well with similarity [Edgar, 2010]. Which means that cluster members are more likely to be found in the first few targets that share the most common unique words. Although this kind of arrangement could increase the speed, but

sequences which fall beyond the assumption might be missed out.

Earlier versions of UCLUST employed CD-HIT definition as the similarity definition, but it uses BLAST identity as the default option since version 6. DNACLUST uses the (semi-)global alignment score while sumaclust by default normalises identity score by alignment length. The identity definition in VSEARCH is customisable to up to five different definitions while the default definition is based on alignment length excluding gaps. Each definition were summarised as table 2.2.

CD-HIT definition	$\frac{\#match}{shorter\_sequence\_length}$
BLAST identity	$\frac{\#match}{alignment\_length + gaps}$
(semi-)global alignment score	$1 - \frac{edit\_distance}{shorter\_sequence\_length}$
VSEARCH identity	$\frac{\#match}{alignment\_length - gaps}$

Table 2.2: Different definitions of sequence identity/similarity

By default, UCLUST does not sort input, but it is customisable to sort by length or abundance. CD-HIT and DNACLUST sort input by length, while *sumaclust* sorts input by abundance because “true sequences should be more abundant” [Mercier et al., 2013]. VSEARCH support both sorting types and the user must specify which one to use.

By default *sumaclust* clusters the input with the "exact" option. The "exact" definition here means that a sequence will be assigned to the cluster of which seed has the most similarity with the sequence, instead of assigning it to the first cluster that was found with the similarity above threshold. DNACLUST flags unclustered sequences within twice the radius from the seed so that they would not be picked up as a cluster centre, but may be included in a cluster with an un-flagged cluster centre [Ghodsi et al., 2011]. This is to prevent the forming of some overlapping clusters with centroids less than twice of the cluster radius.

2.4.3 Swarm – a Single-Linkage Hierarchical Clustering Program

Swarm is an exact, agglomerative, unsupervised, and single-linkage clustering method that produces meaningful OTUs and it has less dependency to the clustering parameters [Mahé, Rognes, Quince, de Vargas, & Dunthorn, 2014]. Swarm was developed to address two major problems that usually happen in the greedy clustering methods, which are:

- Each organism evolved with a different speed while greedy clustering methods use a fixed global threshold that cannot fit both

slow-evolving and rapidly-evolving lineages at the same time.

- As greedy clustering methods are operating according to the input order of amplicons, previous centroids were not re-evaluated as the clustering proceeds which could generate an inaccurate result.

In a comparison study A. May et al. ran 11 different clustering algorithms on some mock dataset which were simulated to mimic raw experimental pyrosequencing data [May et al., 2014]. The outcome of the study showed that despite providing the quality of minimum underestimation when performed on some chimera checked and de-noised data, swarm has a significantly shorter running time compared to other algorithms of a similar quality.

Single linkage is prone to noise data such as a set of vertices forming a long chain connecting supposedly two different clusters, will be identified as one cluster [Jain, Murty, & Flynn, 1999]. This problem is handled in Swarm with a companion algorithm that could identify probable amplicon chains and break them into independent OTUs.

Single-linkage is the only metric that satisfy both SANN (same agglomerative nearest neighbour) property and reducibility property [Olson, 1995]. Suppose there exist cluster i and j to be agglomerated into cluster k, SANN means that nearest neighbours of k is the nearest neighbours that i and j used to have, while reducibility means that k will never be closer to any clusters that i and j were not. Taking advantage of these properties makes it possible to incrementally cluster one item at a time, which gives the benefit of less memory requirement for storing distance values, and less running time [Jain et al., 1999].

Following this property, swarm has the workflow of agglomerating one cluster at a time, and the process is repeated until every amplicon belongs to a cluster. The clustering starts by assigning the first amplicon in the pool as the seed for the first cluster. The seed is then compared against all of the amplicons that remain in the pool. Amplicons that have a similarity above the threshold will be assigned as the subseeds of the cluster and removed from the pool. To get a second layer of subseeds, each subseed will be compared again but only with the amplicons that have no more than $2d$ differences with the seed. This filtering step is based on the triangle inequality that the distance between amplicons that have more than $2d$ differences with the seed cannot have d or fewer differences with the subseed, since the subseed has no more than d difference with the seed. This can be expanded to amplicons that have more than $(k+1)$ differences with the seed cannot have d differences with a k-seed. This process is repeated until no more subseeds can be assigned to the cluster. Then the next amplicon will be assigned as the seed for the second cluster, and the same iterations will be performed until the pool is empty.

2.4.4 Filtering Based on Q-gram/K-mer Distance

Computing similarity between sequences is the core of any clustering algorithm. A sequence belongs to a cluster when the similarity between the sequence and the cluster is above a defined threshold while the definition of similarity may be different between one algorithm and another. For example in the heuristic algorithm a sequence is usually compared to the centroid of the cluster while in hierarchical clustering a sequence is compared to all members of the cluster.

Filtering based on k-mer distance is commonly used in many existing clustering tools seeing that a sequence alignment is expensive in both time and memory. Dynamic programming methods like Needleman-Wunsch algorithm for global alignments and Smith-Waterman algorithm for local alignments both have the time and space complexity of $\mathcal{O}(MN)$ when using the affine gap penalty.

Following definitions from [Ukkonen, 1992], a q-gram or k-mer can be defined as any string $s = a_1 a_2 ... a_k, s \in W^q$ where W^q denotes the set of all permutations of the set $W = \{'A','C','T','G'\}$, the only possible nucleotides in a DNA sequence. Since $|W| = 4$, it follows that $|W^*| = 4^q$.

Let $x = a_1 a_2 ... a_n$ be a string in W^*, if $a_i a_{i+1} ... a_{i+q-1} = v$ for some i, then x has the occurrence of v in x. Let $G(x)[v]$ be the total number of the occurrences of v in x. The q-gram profile of x is the vector $G_q(x) = (G(x)[v])$ for all $v \in W^k$. Let x, y be strings in W^*, the q-gram distance between x and y is

$$D_q(x,y) = \sum_{v \in W^q} |G(x)[v] - G(y)[v]| \tag{2.4}$$

Furthermore, the q-gram distance $D_q(x,y)$ can be evaluated in time $\mathcal{O}(|x|+|y|)$ and in space $\mathcal{O}(|W|^q+|x|+|y|)$. The q-gram distance can be used to predict the edit distance, based on the fact that one edit operation can alter at most q number of q-grams.

2.5 Parallelisation

Parallelisation can be defined as an effort to organise a set of independent subtasks to be executed simultaneously so that the running time of an algorithm can be reduced in a significant way.

There have been several attempts before on designing parallel algorithms for hierarchical clustering algorithms, starting from a parallel algorithm on PRAM proposed by Olson [Olson, 1995], followed by a parallel version of POP (Partially Overlapping Partitioning) algorithm by Dash et al. [Dash, Petrutiu, & Scheuermann, 2004], a parallel algorithm involving MST construction [Olman, Mao, Wu, & Xu, 2009], PARABLE (Parallel Random-Partition Based Hierarchical Clustering) with MapReduce framework [Wang & Dutta, 2011], and other studies that were related to hardware-based approach such as SIMD and optical bus.

Following we highlight several important concepts related to parallel computing [Rauber & Rünger, 2013; Barney et al., 2010].

2.5.1 Levels of Parallelism

There are three levels of parallelism in parallel computing: bit-level parallelism, instruction-level parallelism, and task parallelism. Bit-level parallelism is achieved by increasing processor word size so that single operation will be able to cover a greater size of variables. For example, a 64-bit processor would need only one operation to sum up two 62-bit integers while a 32-bit processor would need two operations.

Instruction-level parallelism means to improve the execution time by increasing number of instructions that can be fetched and executed simultaneously by the processor. Instruction-level parallelism also depends on compiler design and family of the processor where nowadays most modern processors support multiple stages of instruction pipelines.

Task-level parallelism is the parallelism on the programming level – threads programming – where tasks are designed in a way that they can be executed concurrently by multiple processors.

2.5.2 Shared Memory Model

A shared memory model is a memory organisation such that processes share information with each other from a main memory. Examples of shared memory computing model are multithreading and OpenMP for multiple nodes. This model is suitable for programs that require or could benefit from sharing information among its thread. When processes share memory, phenomena like dirty read, phantom read, unrepeatable read, or undefined behaviour might happen, therefore the concurrency must be handled through the synchronisation among threads. There are four different isolation levels that can be implemented when accessing memory, depending on the level of exclusivity that need to be achieved – from lowest to highest:

- concurrent read concurrent write (CRCW)
- concurrent read exclusive write (CREW)
- exclusive read concurrent write (ERCW)
- exclusive read exclusive write (EREW)

2.5.3 Distributed Memory Model

Distributed memory model is the memory organisation such that processes run independently with their own memory and do not share any information with each other. This model is suitable for programs that do not require nor benefit from sharing information among its processes.

A parallel distributed programming model like MapReduce is mainly composed of two steps: *map* and *reduce*. A problem is first partitioned into multiple *maps* that will be solved individually and in parallel on each of the worker nodes. The output of each *map* will then be written on a temporary storage. These outputs will be sorted or shuffled and grouped into a set of *reduce* operations that will be processed by the worker nodes to conclude a final result of the problem.

2.5.4 Estimating Speedup Based on Amdahl's Law and Gustafson's Law

Speedup is a way to measure how much the code improves in running time after being parallelised. It is generally defined as $Speedup = \frac{T_s}{T_p}$ where T_s is time for the sequential program and T_p is time for the parallel version while p is the number of threads. There are two popular laws which estimate the theoretical maximum speedup that one can obtain from parallelising a computation. Amdahl's law observes the speedup based on a fixed problem size, but varying number of processors and the parallel/serial fraction of a parallel program, while Gustafson's law also includes the growing of the problem size into the estimation.

In 1967, Gene Amdahl proposed on Amdahl's law which suggests the expected speedup of a parallel program as:

$$Speedup = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}} \tag{2.5}$$

where s is the portion of the program that cannot be made parallel, p is the portion of the program that can be made parallel and N is the number of processors or threads.

This means that there is a maximum speedup where adding more processors no longer gives more speedup, while the portion of code that can be ' greatly affects how much of maximum speedup [Amdahl, 1967]. Figure 2.4 illustrates the calculated expected speedup as the definition above as more processors were added into the parallel program.

This proposal was revisited in 1988 by John Gustafson who proposed a remark to the Amdahl's law stating that as problem size grows, the speed up of a parallel program would grow linearly [Gustafson, Montry, & Benner, 1988]. Gustafson's law is applicable if the growing of problem size and processor number do not increase the complexity of the serial portion so that the parallel portion will have a runtime of $1/N$ of the serial runtime. Speedup is defined as:

$$Speedup = N - s(N-1) \tag{2.6}$$

where N is the number of processors and s is the serial portion of a program.

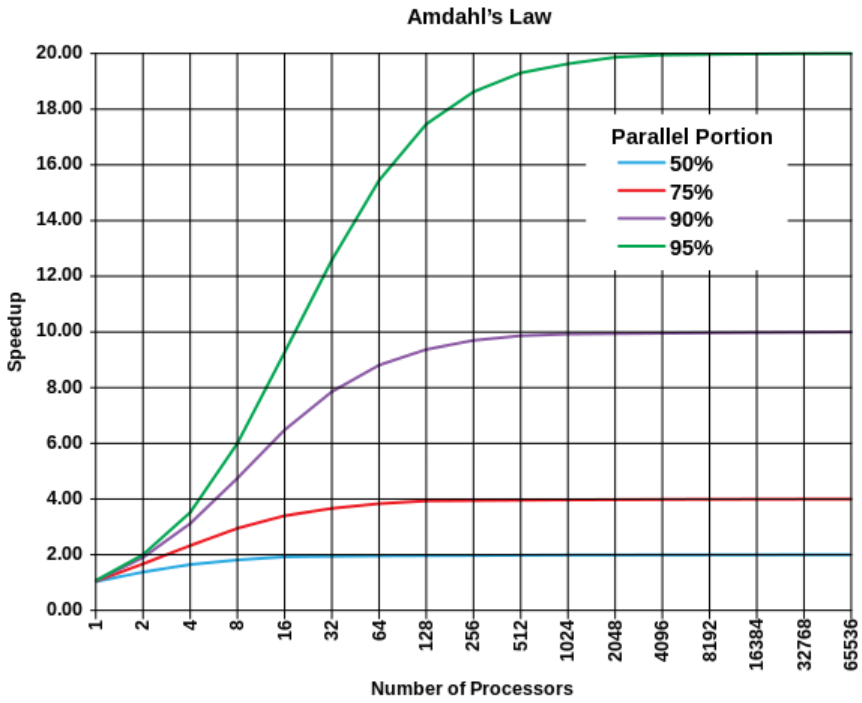


Figure 2.4: Amdahl's Law: Speed up of program using multiple processors in parallel computing
Source: Wikimedia. [2015, January]. Retrieved from http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg

2.6 Summary

Metagenomics has become more popular as the result of the revolution in sequencing technology. DNA database which used to be built upon lab cultivation and human curation now is being replaced by HTS to a limited extent. As a result of the revolution more computing is required to process these data and to eliminate redundant data through clustering so that more costly analysis can be performed on the smaller dataset. Clustering of a sequence dataset could be challenging because sequences belong to an undefined vector space, hence measuring the similarity of two sequences itself is an expensive task. Trends in the past decade show that there is no longer significance improvement in the clock speed of processors. This indicates that parallel programming is necessary to be studied especially on metagenomics clustering where the size of data is growing steadily.

Chapter 3

Design and Algorithms

3.1 Challenges and Limitations

An enormous number of sequences is now available to be processed and this number is still growing steadily. The number of available cores is limited and so is memory space. The aim of the design in this project is to achieve the maximum speedup and scalability through the parallelisation of the sequential algorithm of single-linkage hierarchical clustering, given a limited amount of memory space.

By appreciating the knowledge from Amdahl's law and Gustafson's law, the following concepts stand as the foundation of the design of this project:

- The serial part of the program should be minimised as much as possible.
- The serial part of the program should be designed to grow as slowly as possible when the problem size increases.
- The parallel part of the program should be designed to run as independent as possible so that the overhead would not increase when a number of processors is added.

3.2 Concept of Hierarchical Clustering Algorithm

Hierarchical clustering can be obtained by first calculating the pairwise distance between each vertex followed by constructing a minimum spanning tree that later can be cut on a certain level to get a set of clusters. In single-linkage clustering, the exact distance between each vertex is not so important and would not affect the final result of the clustering since the tree will be cut if the distance between two vertices is greater than a desired resolution. To achieve more efficiency we could take advantage of this attribute by simply storing a boolean type to define the relation between vertices to represent whether they are connected.

Based on the definition above it follows that the number of calculations that needs to be performed on a group of sequences can be defined as the number of possible pairs of two sequences that can be picked from the input sequences. For n input sequences the number of pairwise distances is growing quadratically and can be defined as:

$$C(n,2) = \frac{n!}{(2!(n-2)!)} = \frac{1}{2}(n^2 - n) \tag{3.1}$$

For instance, clustering 10^3 sequences involves $\approx 5 \times 10^5$ pairwise distances while clustering 10^5 sequences would require the computation of $\approx 5 \times 10^9$ pairwise distances. To achieve a better performance some of the calculations can be skipped if they are confirmed as redundant based on the result of previous calculations.

In the course of sequence clustering, deciding the similarity between two sequences requires performing a pairwise global alignment which is quite expensive with complexity of $\mathcal{O}(MN)$ when using the affine gap penalty, where M and N are the length of each sequences. In order to minimise calculation time, unnecessary alignments may be eliminated by a filtering scheme that involves a cheaper computation, such as:

Comparing length of two sequences If two sequences have a length difference more than d then it cannot have the edit distance of less than or equal to d.

Comparing the k-mer profiles of two sequences Sequences that have more than $2dk$ different q-grams or k-mers of length k cannot have the edit distance of less than or equal to d [Mahé et al., 2014]. The length of k affects the cost of the k-mer profiles comparison because the number of possible k-mers $(1-4^k)$ also increases when k is bigger. However the precision that can be obtained by using a different value of k remains open.

Figure 3.1 illustrates the 3-mer vector comparison of two sequences: ACTACGTGAAG and ACTACGTAAAG that we know have one single different nucleotide. This simulation follows the design in Swarm, where a q-gram profile vector contains either 0 or 1 to represent even or odd occurrences of a q-gram in the sequence. A more common practice is to assign 0 for no occurrence of the q-gram, and 1 for any occurrence(s) of the q-gram.

The first step of the comparison would be to generate the q-gram profile vectors for both sequences that register if the sequence has the odd number or the even number of 3-mer occurrences. Since there are four nucleotides A, C, T, G in a DNA sequence, then this vector has the length of 4^3 with value 0 for every even number of 3-mer occurrences, and value 1 for every odd number of 3-mer occurrences. No occurrence is registered as an even number of occurrences. Q-gram/k-mer distance of the two sequences can later be obtained by summing up the XOR result of each vector members. Two sequences with a k-mer distance more than $2dk$ cannot have less than or equal to d of edit distance. In

this example, the k-mer distance is not more than $2dk = 2(1)(3) = 6$ which means that the two sequences may have the edit distance of less than or equal to 1.

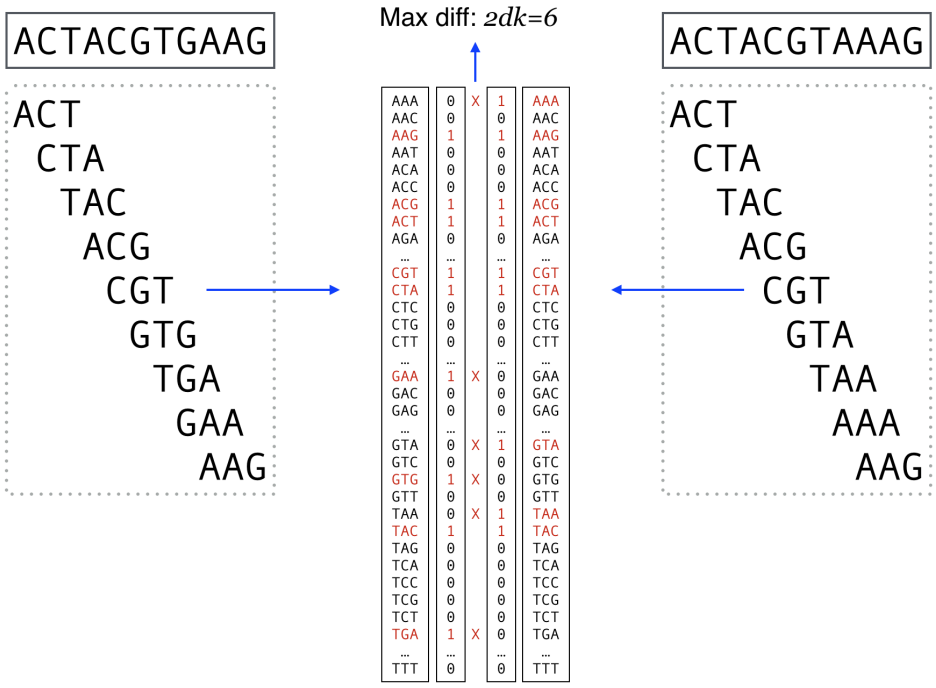


Figure 3.1: Simulation of 3-mer vector comparison of two sequences

We have tested the following approaches during the development of this project, i.e.: (i) comparing the length of sequences and then proceed with comparing k-mer profiles, or (ii) only comparing k-mer profiles. As the result, approach (i) did not give a meaningful time improvement comparing to approach (ii), while in the algorithm of this project the comparison of k-mer profiles is always necessary for the economic search (see subsection 3.5.1.2). Therefore to rule out unnecessary alignments only the k-mer profiles comparison will be used in this project. K-mer profiles for all sequences are to be pre-calculated during the initialisation of the program and stored in the RAM so that the cost to compute the k-mer distance during the clustering is $\mathcal{O}(1)$.

3.3 Parallelisation Strategies

There are a few approaches to the parallelisation that were explored during this work in order to find out a possible way to utilise multiple threads. The first approach that will be explained in section 3.4 is a distributed memory model where multiple parallel processes will each carry out an individual clustering on the partitions of sequences. Finally, a serial process will merge the results to conclude a result set for the entire dataset. The second approach in section 3.5 is a shared

memory model where the entire clustering process will be divided into a set of small tasks – referred as "row calculations" in the design – that can be carried out independently across multiple threads.

3.4 Distributed Memory Model: Parallel Clustering on Partitions of Dataset and Merging of Results

Sequences were partitioned into some groups that become the input sequences for each thread. Each thread would simultaneously cluster the input sequences using the same agglomerative algorithm from Swarm, producing their own result sets. Afterwards, these result sets can be merged either serially or in parallel to conclude a final answer for all of the sequences.

Let t be the number of parallel threads and N be the total input sequences, each partition of the distributed model would have $\frac{1}{t}N$ number of input sequences. Since the complexity of a serial hierarchical clustering is $\mathcal{O}(N^2)$, it means that the parallel clustering of each partition would take only $\frac{1}{t^2}T$, where T is the time to cluster an N number of sequences.

After each partition of the dataset has been individually clustered in each thread, the next step would be to merge the result sets from each thread into a single result set. Let clusters from first clusters set be $A = \{A_1, A_2, ..., A_m\}$ and clusters from second clusters set be $B = \{B_1, B_2, ..., B_n\}$. For each member of the Cartesian product $A \times B = \{(a, b) | a \in A, b \in B\}$ where $|A \times B| = |A| \cdot |B|$ it shall be evaluated whether the pair should belong to the same cluster. If yes, then they will be merged. Once all combinations have been observed, all members of A and B will be put together as a single result set.

The following steps describe the process of the evaluation whether a pair of clusters should be merged: Let the first cluster be A_1 and let the second cluster be B_1, if there exists any pair of sequences where $d(a, b) \leq d, a \in A_1, b \in B_1$ then A_1 and B_1 belongs to the same cluster. In other words, two clusters should be merged if there exists any sequence in the first set that has the distance less than or equal to the clustering resolution with any sequence in the second set.

The following steps describe the shortcut for verifying that two clusters cannot be merged: Let S_a be the seed of the first cluster, S_b be the seed of the second cluster, and $d(S_a + S_b)$ be the edit distance between both sequences as illustrated on figure 3.2. Let D_a be the maximum level of the subseeds from the first cluster and D_b be the maximum level of the subseeds from the second cluster. If $d(S_a + S_b) > D_a + D_b$, then there cannot exist any sequences from each first and second clusters that are near with each other. On the contrary $d(S_a + S_b) \leq D_a + D_b$ does not mean the opposite, because the shape of a cluster is high dimensional and the variables in this equation do not belong to a metric space.

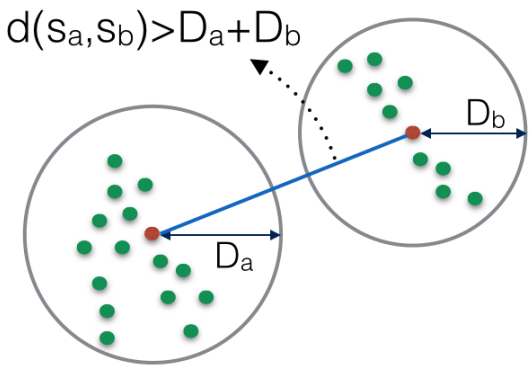


Figure 3.2: Example of N sequences pairwise matrix

The result sets from each thread shall be merged with each other to a single clusters set. Up to this point the result is not yet final because whenever a cluster was merged with another cluster, it changes the structure of the cluster in a way that it might be connected with a cluster that was previously not merge-able. Therefore it is necessary to perform a recursive process as the last step that will compare these "merged clusters" with every other cluster in the clusters set until it is confirmed that there is no longer any merge-able clusters.

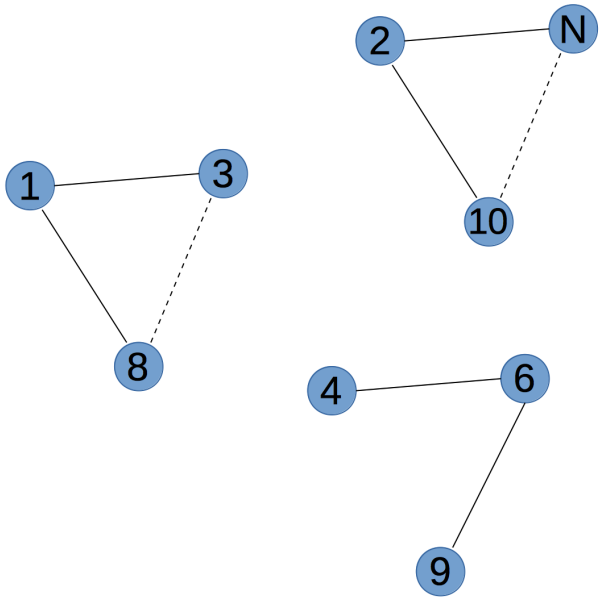
A set of code were written as a proof of concept to verify this idea, and it was run on a test dataset of around 1,500 sequences. The result shows that the part of the algorithm which merge the result sets from each parallel process was very inefficient because there were too many pairs of clusters that need to be evaluated especially in a natural biological dataset. Therefore, the work for this strategy was discontinued because of the high complexity of the merging problem.

3.5 Shared Memory Model: Parallel Calculations of Pairwise Distances

Given a set of input sequences $S = \{1,2,3,...,N\}$, hierarchical clustering can be obtained by calculating pairwise distances of each pair combinations of the sequences $C = \{\{1,2\},\{1,3\},...,\{1,N\},\{2,1\},\{2,2\},...,\{N,N\}\}$. The complete comparison of all pairs that was mentioned before will be denoted in a form of "pairwise connection matrix" or "pairwise matrix" as shown on figure 3.3. We found this matrix to be useful as the thinking aid for breaking down the complete step of hierarchical clustering into small tasks so that they can be run in parallel.

	1	2	3	4	5	6	7	8	9	10	...	N
1	-	0	1	0	0	0	0	1	0	0		0
2	-	-	0	0	0	0	0	0	0	1		1
3	-	-	-	0	0	0	0	0	0	0		0
4	-	-	-	-	0	1	0	0	0	0		0
5	-	-	-	-	-	0	0	0	0	0		0
6	-	-	-	-	-	-	0	0	1	0		0
7	-	-	-	-	-	-	-	0	0	0		0
8	-	-	-	-	-	-	-	-	0	0		0
9	-	-	-	-	-	-	-	-	-	0		0
10	-	-	-	-	-	-	-	-	-	-		1
...												
N	-	-	-	-	-	-	-	-	-	-		-

Figure 3.3: Example of N sequences pairwise matrix



Note: The dashed line between node 3 and 8 denotes a redundant calculation for $c(3,8)$ because $c(1,3) = 1$ and $c(1,8) = 1$ from earlier comparisons. And so is $c(10,N)$.

Figure 3.4: Sequence nodes correlated to pairwise matrix in previous figure.

Each cell in the matrix contains the information whether a sequence is connected to another sequence, $c(a,b) = 0$ means that a and b are not connected while $c(a,b) = 1$ means that a and b are connected. Two nodes are connected if they have the difference of less than or equal to a specified clustering resolution d. The matrix can be used to build a set

of dendrograms (figure 3.4) that will lead to the final clustering result. A single row of the matrix represents the work of comparing a sequence to every other sequences with an ID greater than itself, which will be referred as "row calculation" in this paper, i.e., the task of calculating the distances between the row ID and all of the column IDs.

A pairwise matrix is always symmetrical since $d(a,b) = d(b,a)$. Furthermore $d(a,a) = 0$, therefore the values on the main diagonal do not need to be computed. This means that a pairwise matrix consists of indeed $\frac{1}{2}(n^2 - n)$ meaningful cells. Given a set of input sequences to be clustered $S = 1,2,3,...,10$, based on the pairwise matrix as presented in figure 3.4, a final clustering result set would be concluded as: $R = \{\{1,3,8\},\{2,10\},\{4,6,9\},\{5\},\{7\}\}$.

The calculation on each cell in the matrix as well as each row is independent and do not need to be chronological in order to get a correct result. With this kind of arrangement, each "row calculation" can be performed simultaneously across multiple threads. A complete hierarchical clustering process of N input sequences consists of $(N - 1)$ row calculations. While each row calculation consists of $(N - m)$ pairwise similarity comparisons, where m is the ID of the sequence in the input set ordered decreasingly by abundance. In other words, the row calculation for the first sequence would have $(N - 1)$ pairwise similarity comparisons while the one for the last sequence would have zero comparison.

The parallelisation in this proposal is to create a thread pool consists of t number of threads. A thread organiser (or the main program) keeps track of the current row ID in the pairwise matrix computation. Whenever there is a thread that becomes available, it will be assigned to perform a row calculation for the current row ID. The thread organiser will then increment the current row ID and proceed to assign a row calculation for another idle thread. Once the row ID has reached the total number of input sequences, the process is considered as done, and all threads will be destroyed.

3.5.1 Omitting Redundant Calculation

The omitting redundant calculation is necessary especially for a large set of data and may lead to substantial performance improvement.

3.5.1.1 Omitting Edges Outside of Minimum Spanning Tree

The computation of the pairwise distance between a and b will be redundant if it was already known from earlier calculations that a and b belong to the same cluster. For instance in the previous figure 3.4, node 1, 3 and 8 all belong to the same cluster. Since $d(1,3) = 1$ and $d(1,8) = 1$ are already sufficient to prove that $\{1,3,8\}$ are a cluster, it would be redundant to calculate the pairwise distance between node 3 and node 8. Calculations that are similar to this example can be safely

omitted while they do not compromise the precision of the final result if they were computed anyway.

3.5.1.2 Economic Search on First Level Connections

Given the condition where node a is connected with each node in $Z = \{b_1, b_2, ..., b_n\}$ or $c(a,b) = 1, b \in Z$ and the distance between a and every node in $X = \{x_1, x_2, ..., x_n\}$ are more than twice the clustering resolution i.e. $d(a,x) > 2d, x \in X$, as shown on figure 3.5. Then the calculation of pairwise distances $d(b,x), b \in Z, x \in X$ can be safely omitted since they cannot be less than or equal to the resolution. This is because $d(a,x) \leq d(a,b)+d(b,x), b \in Z, x \in X$ and therefore $d(b,x) \geq d(a,x)\text{-}d(a,b), b \in Z, x \in X$.

In another word, it is sufficient to compare $Z = \{b_1, b_2, ..., b_n\}$ with $E = \{e_1, e_2, ..., e_n\}$ where $\min_{e \in E} d(a,e) \leq 2d$.

The concept of economic search here is defined by following the strategy in Swarm where a subseed would only be compared to the amplicons which have no more than $2d$ differences with the seed [Mahé et al., 2014]. In Swarm an identical economic search is performed without any limit to the level of connection, and therefore it may be more efficient than the economic search in this project. The limiting of the level of connection in this project is inevitable in order to keep the row calculations independent.

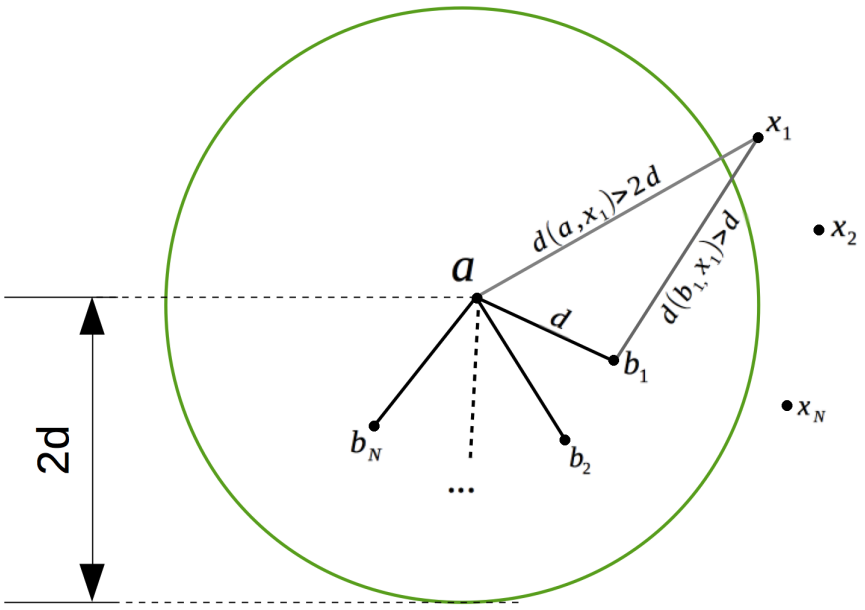


Figure 3.5: Economic search on first level connections.

3.5.1.3 Economic Search on Multiple Levels Connections

On the previous explanation, it was described how to skip the distance calculation on the set of first level connected sequences. The similar strategy may be further extended to skip the distance calculations on

the r_{th} level of connected sequences if the distance of every sequence with the row sequence will be stored temporarily in the memory. On the first level connected sequences only the sequences with the distance no more than twice the resolution should be considered as candidates, and on the r_{th} level of connected sequences only sequences within $(r+1)d$ needs to be considered, where d is the resolution.

In other words, let a be a sequence and $Z_1 = \{b_1, b_2, ...b_n\}, \min_{b \in Z_1} d(a,b) \leq d$. Let $\min_{a \in Z_1, b \in Z_2} d(a,b) \leq d$, and furthermore $\min_{a \in Z_{r-1}, b \in Z_r} d(a,b) \leq d$, then it is sufficient to compare $Z_r = \{b_1, b_2, ..., b_n\}$ with $E = \{e_1, e_2, ..., e_n\}$ where $\min_{e \in E} d(a,e) \leq (r+1)d$.

3.5.2 Forming Clusters Set

Building clusters based on the information from a pairwise matrix is rather simple:

- Iterate on every connection record in the pairwise matrix. Whenever a connection (x, y) exists, perform a lookup on clusters set to see if x and y belong to any clusters.
  - If x and y belong to a same cluster, nothing needs to be done.
  - If x and y belong to two different clusters, then merge these clusters.
  - If x belongs to a cluster while y is new (does not belong to any cluster), then add y to the cluster. And vice versa.
  - If x and y are both new, create a new cluster and add both x and y to the new cluster.
- Upon completing the calculation of the pairwise matrix, there might exist some sequences that do not belong to any cluster. These sequences need to be identified and individually added as singleton clusters as the final step.

3.6 Data Structure Design

The data structure design is important since the problem size could be as big as 40 GB input of fasta file, or even more. The memory requirement for the program grows together with the problem size, meanwhile the amount of maximum memory is limited by the current available hardware on the market, operating system, or simply the budget of the user.

3.6.1 Storing Cluster Sets

A cluster consists of a cluster ID and a set of cluster members. The clusters set will be stored as a mapping from the cluster ID to its cluster members. In this way, any cluster can be retrieved efficiently based on its cluster ID. Following the step to form a cluster as described above,

it is also necessary to store the reversed mapping information from a sequence member to its cluster ID. This is to facilitate the lookup of checking whether a sequence x belongs to any cluster.

3.6.2 Storing of Pairwise matrix

The pairwise matrix data structure that was described earlier could take up a lot of space in memory. For an N number of sequences and a boolean type or bit type for each of the matrix cell, it would take up to N^2 bits to store the pairwise matrix. It means that 10^6 sequences require 10^{12} bit = 125 GB of memory, which is impractical.

Saving the connection matrix as a dynamic list of connected pair of sequences can save up a lot of space because the connection matrix is usually a sparse matrix with more cells containing zero values. However, the lookup on a list is usually as slow as $\mathcal{O}(N)$ or $\mathcal{O}(logN)$ for a sorted list but this would not affect the execution time since our algorithm does not require any lookup on the pairwise matrix data.

To save more time and memory, there is no need to store the connection pairs if the step of building cluster sets can be done on the fly as a connection was detected. This is feasible since the way of forming clusters set was designed to be independent and the order of inserting pairs of connected sequences into the clusters set does not affect the clustering result. A synchronisation effort is necessary to prevent concurrent read/write phenomena on the global clusters set data. Only one thread at a time should be allowed to perform a lookup for any sequence on the clusters set, otherwise some out of date information could be retrieved and might lead to a wrong result set. For this reason, there will be a small overhead to synchronise the read/write access to the global cluster set data.

3.6.3 Storing Flags for Economic Search

As explained before, some calculations are redundant based on the triangle inequalities of the distance between three sequences and they may be eliminated without affecting the final result of the clustering. The economic search of first level connection can be achieved by storing a set of sequences that should be considered as the candidates for the row calculations of all connected sequences.

While for multiple level economic search, the distances from initial seed to all candidates must be stored in addition to the sequence itself so that these candidates can be grouped according to their level of connections with the row sequence. To minimise overhead, each thread will have their own temporary variables to store this information.

3.7 Implementation

3.7.1 Main Program

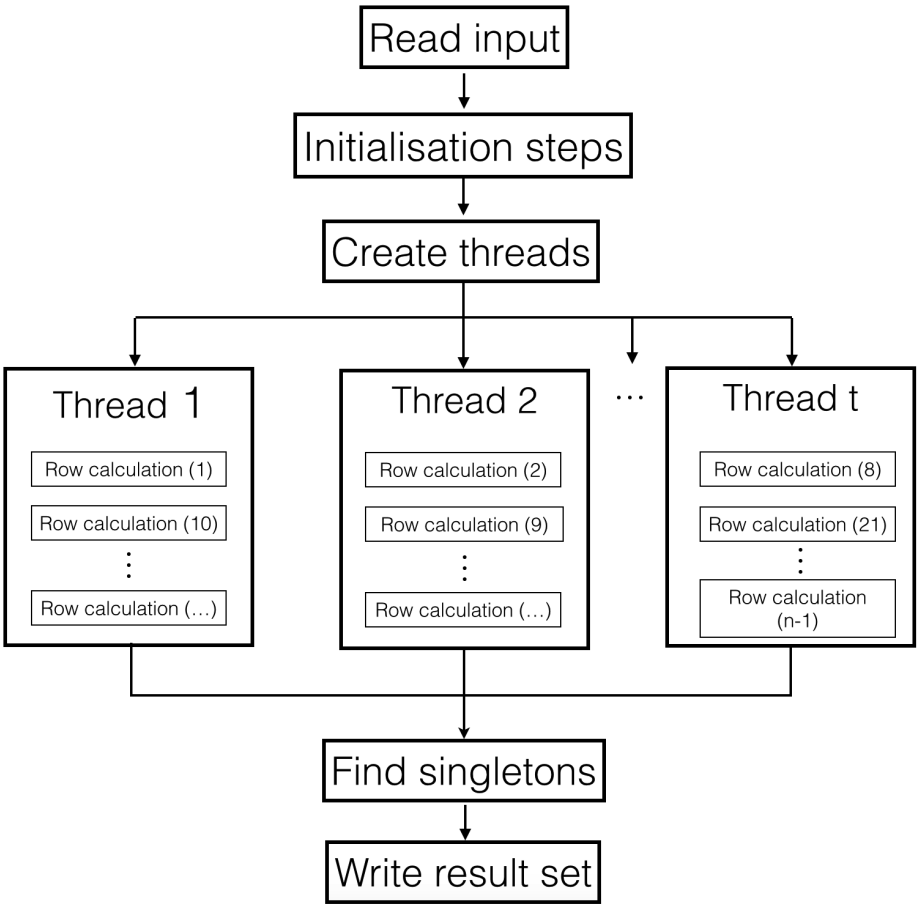


Figure 3.6: Flowchart of the whole process.

The main program starts by receiving a FASTA file, the value of d, the depth of economic search, and the number of threads as the input. A preliminary check consists of the following steps:

- validation that there is no duplicate sequence header in the input file.
- validation that all sequences are comprised of valid nucleotides $\{A, C, T, G\}$.
- validation that all sequences contain an abundance information and are sorted decreasingly by this abundance information.

Input sequences are stored in the global memory and the following attributes are generated for each sequence:

- sequence index, based on the order of the input, e.g. the first sequence has the index = 0, the second sequence has the index = 1, and so on.

- q-gram or k-mer profile (as described on section 2.4.4 and 3.2)
- length of the sequence
- visit flag, pre-defined as 0 (means not visited)

Followed by the initialisation of t number of threads, each idle thread will be assigned with a row ID starting from 1 until (N-1) where N is the number of input sequences. Based on the submitted "depth" option, a brute-force approach, or a *first-level* economic search, or an n-level/*multiple-level* economic search may be performed inside the thread for the designated row ID. By default, the depth of the economic search is 1, which also means that no economic search will be performed, or a brute-force row calculation will be used. For $depth = 2$ it means that the economic search will be performed within $2d$ radius from the seed, or referred as *first-level* economic search. For $depth > 2$ it means that the economic search will be performed within $(depth)d$ radius from the seed, or referred as a *multiple-level* economic search.

The single unit of work that is performed by each thread is referred as the "row calculation". The operation for each method will be explained separately in the following subsections. The assigning of each row calculation to an idle thread shall go on until the row ID reaches (N-1) where the parallel work is considered as done and all threads may be destroyed. Up to this point, the construction of the clusters set is not yet final because some sequences that are not connected with any other sequences – referred as the singletons– were not included in the clusters set. A check will be performed for all sequences to verify whether the sequence belongs to any cluster, if not then a new cluster will be created for the sequence.

The flowchart of the entire process can be seen on figure 3.6.

Algorithm 1 Main Process

Input:
T : thread count
$file$: input FASTA file name and path
d : clustering resolution
$depth$: the level for economic search

Output:
$\mathbb{C} = \{C_1, C_2, ...C_m\}$ ► clusters set

preconditions checking on submitted file
load all sequences S and generate k-mer profiles K, save S and K into main memory
$N \leftarrow$ number of sequences
initialise and start all threads T
for $row \leftarrow 1$ to N **do**
    wait for available thread
    if $depth = 1$ **then**
        Call procedure brute_force_row_calculation (row, N, d, s, k)
    else if $depth = 2$ **then**
        Call procedure first_level_row_calculation (row, N, d, s, k)
    else
        Call procedure multiple_level_row_calculation (row, N, d, s, k, depth)
    end if
end for
wait until all threads are finished
terminate and destroy all threads
for $seq \leftarrow 1$ to N **do**
    if $seq \notin \mathbb{C}$ **then** ► if sequence was not in any cluster
        $C_{m+1} \leftarrow seq$ ► create new cluster and assign sequence
        $\mathbb{C} \leftarrow \mathbb{C} \cup C_{m+1}$ ► add cluster to clusters set
    end if
end for
return $\mathbb{C}$

3.7.2 Row Calculation

Row calculations are individually performed inside each thread, one row calculation can be referred as one work unit where a thread may perform more than one work units in the entire clustering process. There are three options of row calculations that were defined for our program: brute-force search, first-level economic search, and multiple-level economic search.

3.7.2.1 Brute-Force Search

A brute-force search does not involve any skipping mechanism and it will simply go through all sequences from $m+1, m+2, ...N$ to identify a

set of connected sequences, where m is the row ID and N is the total input sequences. A connected sequence is a sequence that has the edit distance less than or equals to d with the row ID. To check whether a sequence is connected, the k-mer profile of both sequences will be compared. If there are more than $2dk$ differences then it means that the sequence is not connected, otherwise it will proceed with a pairwise alignment to verify if the edit distance is less than or equal to d. Every time a connection was identified, algorithm 3 will be called to assign the pair of connected sequences into the correct cluster.

Algorithm 2 Row Calculation for Brute-Force Search Method

Procedure: brute_force_row_calculation (row, N, d, s, k)
Input:
row : index of the row sequence
N : total number of input sequences
d : clustering resolution
S : all input Sequences
K : K-mer profiles of all input sequences

1: **for** $col \leftarrow row+1$ to N **do**
2: **if** difference(K_{row}, K_{col}) $\leq d$ **then**
3: ▶ K_n is the k-mer profile of sequence S_n
4: **if** alignment(S_{row}, S_{col}) $\leq d$ **then**
5: FORM_CLUSTER(row,col)
6: **end if**
7: **end if**
8: **end for**

3.7.2.2 Inserting Sequences into the Clusters Set

This procedure is called by submitting x and y as the arguments, where x and y are a pair of sequences which were identified to be connected or similar. Both sequences x and y or one of them will be inserted into the clusters set based on a set of rules described in section 3.5.2.

The global clusters set are denoted as $\mathbb{C} = C_1, C_2, ..., C_n$ in the pseudocode 3, where $C_x = x_1, x_2, ..., x_m$ is a cluster consisting a set of m connected sequences. By the end of the clustering task, $\mathbb{C}$ would contain all of the input sequences grouped into n clusters. A singleton cluster is a cluster that contains only one member.

As mentioned in section 3.6.1, there are two maps that are used to store the clusters set information. The first map is implemented using an unordered map from Boost library as the container: `boost unordered_map<unsigned long int, cluster_info>`. It contains the mappings from the cluster ID to a `cluster_info` object. The `cluster_info` is a structure that consists of the cluster ID, and a list of sequences IDs of the cluster members (`std::vector<unsigned long int>`).

The second map is implemented as a primitive array `unsigned long int * member_stats`, where the array index represents the sequence ID and the value of the array represents the cluster ID. The reason to use a primitive array is to minimise the overhead based on the fact that the length of the array is fixed to the total input sequences.

To add a sequence to a cluster, the sequence ID is appended to the list of cluster members in the `cluster_info` followed by assigning `member_stats[sequence_id] = cluster_id`. The second map `member_stats` provides a fast access for checking whether a sequence is a member of a certain cluster.

The access to both of the maps must be protected by a lock since they are accessible to all of the threads and concurrent read or write operations on these objects may lead to the phenomenon of reading outdated information and undefined behaviour. The lock is obtained in the start of the update cluster procedure and released before exiting the procedure.

Algorithm 3 Assigning Sequences into the Clusters Set

Procedure: form_clusters (x, y, $\mathbb{C}$)
Input:
x : index of the first sequence
y : index of the second sequence
$\mathbb{C}$: the global clusters set

1: wait and obtain lock for $\mathbb{C}$
2: **if** $x \in \mathbb{C}$ and $y \in \mathbb{C}$ **then** ▶ Merge second cluster into first cluster
3: $C_x \leftarrow C_x \cup C_y$ where $C_x : x \in C_x, C_y : y \in C_y$
4: $\mathbb{C} \leftarrow \mathbb{C} - C_y$
5: **else if** $x \notin \mathbb{C}$ and $y \notin \mathbb{C}$ **then**
6: $C_{m+1} \leftarrow \{x, y\}$ ▶ Create new cluster for both sequences
7: $\mathbb{C} \leftarrow \mathbb{C} \cup C_{m+1}$
8: **else if** $x \notin \mathbb{C}$ and $y \in \mathbb{C}$ **then**
9: $C_y \leftarrow C_y \cup x$ where $C_y : y \in C_x$
10: **else if** $x \in \mathbb{C}$ and $y \notin \mathbb{C}$ **then**
11: $C_x \leftarrow C_x \cup y$ where $C_x : x \in C_x$
12: **end if**
13: release lock for $\mathbb{C}$

3.7.2.3 First Level Connection Economic Search

A first level connection economic search includes the same step as a brute-force search with the extra work to save two groups of sequences in a temporary variable for the subsequent calculations, i.e. first level connected sequences, and candidates for the calculation of these sequences. In this option, one work unit of a thread consists of calculation of the row itself and row calculations of first level connected sequences. For instance if the row sequence n is connected with Z =

$\{b_1, b_2, ...b_m\}$, and $E = \{e_1, e_2, ..., e_p\}$ is a set of candidate sequences that each has distance not more than $2d$ from n, then a row calculation consists of the computation for rows: n and Z, but sequences in Z will only be compared to candidate sequences E that have the ID greater than itself.

The flag for identifying whether row n has been visited is denoted as B_n in the pseudocode. The flag is necessary to prevent a row from being visited more than once because one row calculation may involve several non-sequential rows. Before performing a row calculation the procedure checks whether the flag for the row ID was already set as "visited". If yes, then it will exit the procedure because the row has been visited before, otherwise the flag will be updated to “1” and the procedure proceeds to the row calculation.

Algorithm 4 Row Calculation for First Level Connection Economic Search Method

Procedure: first_level_row_calculation (row, N, d, s, k)

Input:

row : index of the row sequence
N : total number of input sequences
d : clustering resolution
S : all input Sequences
K : K-mer profiles of all input sequences

if $B_{row} = true$ **then**
    return ▶ Exit this procedure because row has been visited before
else
    update $B_{row} \leftarrow true$
end if
for $col \leftarrow row + 1$ to N **do**
    $kmer\_diff \leftarrow$ difference(K_{row}, K_{col})
    if $kmer\_diff \leq d$ **then**
        $edit\_distance \leftarrow$ alignment(S_{row}, S_{col})
        if $edit\_distance \leq d$ **then**
            FORM_CLUSTER(row,col)
            $Z \leftarrow Z \cup col$ ▶ Z is the set of all sequences connected to row
        else if $edit\_distance \leq 2d$ **then**
            $E \leftarrow E \cup col$ ▶ E is the set of all candidates for Z
        end if
    else if $kmer\_diff \leq 2d$ **then**
        $E \leftarrow E \cup col$
    end if
end for
for all $seq \in Z$ **do**
    if $B_{seq} = false$ **then**
        update $B_{seq} \leftarrow true$
        for all $cand \in E$ such that $cand > seq$ **do**
            ▶ Only need to consider candidates with ID greater than the sequence
            if difference(K_{seq}, K_{cand}) $\leq d$ **then**
                if alignment(S_{seq}, S_{cand}) $\leq d$ **then**
                    FORM_CLUSTER(row,col)
                end if
            end if
        end for
    end if
end for

3.7.2.4 Multiple Level Connection Economic Search

A multiple level connection economic search is the extension of previous procedure where an economic search of ($depth \geq 2$) levels is to be performed, where $depth$ is the desired level of economic search. For each level, two groups of sequences: (i) a set of connected sequences, (ii) a set of candidate sequences, will be stored in the temporary memory.

Let connected sequences $\mathbb{Z} = \{Z_1, Z_2, ..., Z_{depth}\}$ and candidates sequences $\mathbb{E} = \{E_1, E_2, ..., E_{depth}\}$ be as defined in section 3.5.1.3. Then one row calculation consists of the computation for rows: n and $\mathbb{Z}\}$, but sequences in Z_1 will only be compared to candidate sequences E_1 that have the IDs greater than itself, Z_2 with E_2 and so on. Furthermore, the sequences in E_1 that were not connected with any of the sequences in Z_1 will be included in E_2 as the candidates for Z_2, and so on.

Algorithm 5 Row Calculation for Multiple Level Connection Economic Search Method

Procedure: multiple_level_row_calculation (row, N, d, s, k, depth)
Input:
row : index of the row sequence
N : total number of input sequences
d : clustering resolution
S : all input Sequences
K : K-mer profiles of all input sequences
$depth$: maximum level of economic search

1: **if** $B_{row} = true$ **then**
2: **return**
3: **else**
4: update $B_{row} \leftarrow true$
5: **end if**
6: **for** $col \leftarrow row + 1$ to N **do**
7: $kmer\_diff \leftarrow$ difference(K_{row}, K_{col})
8: **if** $kmer\_diff \leq d$ **then**
9: $edit\_distance \leftarrow$ alignment(S_{row}, S_{col})
10: **if** $edit\_distance \leq d$ **then**
11: FORM_CLUSTER(row,col)
12: $Z_1 \leftarrow Z_1 \cup col$ ▶ Z_1 is the set of sequences connected to row
13: **else if** $edit\_distance \leq (depth+1)d$ **then**
14: $E_1 \leftarrow E_1 \cup col$ ▶ E_1 is the set of all candidates for Z_1
15: **end if**
16: **else if** $kmer\_diff \leq (depth+1)d$ **then**
17: $E_n \leftarrow E_n \cup col$ where $(n)d < d(row, col) \leq (n+1)d$
18: **end if**
19: **end for**

Row Calculation for Multiple Level Connection Economic Search Method (continued)

```
20: for r ← 2 to depth do
21:     for all seq ∈ Z_r do
22:         if B_seq = false then
23:             update B_seq ← true
24:             for all cand ∈ E_r such that cand > seq do
25:                 if difference(K_seq, K_cand) ≤ d then
26:                     if alignment(S_seq, S_cand) ≤ d then
27:                         FORM_CLUSTER(row,col)
28:                         Z_{r+1} ← Z_{r+1} ∪ col
29:                             ► Z_{r+1} is the set of sequences connected to Z_r
30:                     else
31:                         E_{r+1} ← E_{r+1} ∪ col
32:                          ► Candidates that are not close enough for this
   level should be added to next level
33:                     end if
34:                 else
35:                     E_{r+1} ← E_{r+1} ∪ col
36:                 end if
37:             end for
38:         end if
39:     end for
40: end for
```

Chapter 4

Methods and Materials

4.1 Programming Language and Libraries

P-swarm was developed in C++ version 98 and compiled with GCC – the GNU compiler collection. C++ is an extension to C and it has the same syntax as C. It is a multi-paradigm language which supports object oriented programming while does not enforce it. C++ comes with a set of standard library which consists of many useful implementations such as the container classes (`vector`, `map`, `list`, `queue`, etc).

Multithreading in p-swarm was implemented using POSIX thread (pthread) and `mutex` for shared memory mutual exclusion. Some codes for k-mer comparisons and global pairwise alignment that are very efficient and fast were reused from Swarm, with some efforts to migrate them to C++.

As the unit testint, the result set of p-swarm was compared with the result from Swarm to ensure that the program gives the correct results for different resolutions and numbers of threads.

4.2 Development Tools

4.2.1 Eclipse IDE for C/C++ Developers

Eclipse IDE has rich features for software development, such as: syntax checking, refactoring tools, code formatting, project and build, source code navigation, static code analysis and so on.

4.2.2 GDB for Program Debugging

GDB (GNU project debugger) is a debugging tool that can show us what the program was doing at the moment it crashed. GDB can be used to debug several types of languages such as Ada, Pascal, C, C++, Objective-C, and so on.

4.2.3 Valgrind for Memory Analysis

Valgrind is a dynamic analysis tool that provides a number of debugging and profiling tools. During the development of this project, the following tools from Valgrind were utilised to detect memory-related errors and bugs.

4.2.3.1 memcheck

Memcheck is the memory leak detector. This tool can generate a detailed report whenever memory leaks happen and it can also detect uses of uninitialised values. The stack trace generated by memcheck can tell us where the leaked memory was allocated and which kind of leaks happened, e.g. "definitely lost", "indirectly lost", or "possibly lost".

4.2.3.2 massif

Massif was used in this project to measure the amount of heap memory was consumed during runtime. Since C/C++ does not come with an automatic garbage collector, this tool could be very helpful in reducing the memory consumption and to pinpoint location of code with bad design that did not properly clean up object in the memory.

This tool records the snapshots of heap memory usage at every unit of time (measured by numbers of instructions executed) and it can also specify the type and originator of the objects that were occupying the heap.

Massif is usually being used together with `ms_print` that will aggregate the snapshots that were written by `massif` and summarise them into some tables and graphs that represent the growth of the heap across the entire timeline. A normal graph should show ideal growth of heap where the dead objects did not pile up.

4.2.4 Instruments from XCode for Profiling

Instruments is a performance analysis tool that was bundled as a part of Apple's XCode development tools. It can be used to dynamically trace and profile OS X and iOS code, but can also be used to analyse C/C++ code in general. This project was using the time profiler template from Instruments in order to gain a deeper understanding on the workload and was useful to pinpoint the part of the code that was performing slow.

4.3 Hardware for Testing

Testings were performed on the Abel computer cluster – a shared resource for high-performance computing at the University of Oslo, hosted by USIT RIS group. Abel is comprised of more than 650 compute nodes each with minimum 64 GB RAM, 16 physical CPU cores – dual

Intel E5-2670 (Sandy Bridge) based running at 2.6 GHz. There are a few compute nodes that are equipped with up to 1 TB RAM, 32 physical CPU core – Intel E-4620 based running at 2.20 GHz[1].

All compute nodes are running Linux, 64 bit of CentOS 6. The queue system in Abel is managed by SLURM workload manager.

4.4 Dataset for Testing

To observe the program's performance, the tests were conducted on several datasets with an assorted number of amplicons and average nucleotides length as shown on table 4.1.

Name	raw reads	unique raw reads	unique amplicons	average nucleotides
EMP [i]	1.44×10^{11}	1.28×10^{9}	1.55×10^{8}	117
BioMarKs [ii]	5.63×10^{8}	1.19×10^{8}	3.12×10^{5}	381
Tara [iii]	5.74×10^{10}	1.20×10^{9}	9.50×10^{6}	129
Greengenes [iv]	1.67×10^{9}	1.33×10^{9}	9.48×10^{5}	1,404

[i] Sample obtained from 60 studies of EMP project [Gilbert, Jansson, & Knight, 2014].
[ii] Sample obtained from BioMarKs project [Logares et al., 2014].
[iii] Sample obtained from TARA OCEANS project [Karsenti et al., 2011].
[iv] Sample obtained from Greengenes database[DeSantis et al., 2006] version 13.5.

Table 4.1: Summary of Test Datasets

Sample processing, sequencing and core amplicon data analysis for EMP.fas were performed by the Earth Microbiome Project (www.earthmicrobiome.org) and all amplicon and metadata has been made public through the data portal (www.microbio.me/emp).

4.4.1 Subsampling

The massive dataset such as EMP.fas will be sampled randomly to select a subset of the dataset as the representative. We are using *vsearch* [Flouri et al., 2015][2] for this purpose, for example with the following command:

```
vsearch --subsample EMP.fas --fraction 0.1 --output EMP_0.1.fas
--seed 0
```

Running above's command would randomly select 10% of the original sequence read **before dereplication**, and thus the result of the subsampling would not be comprised of exactly 10% of the entire unique sequences that could represent a more natural partition of a dataset's subset.

[1] http://www.uio.no/english/services/it/research/hpc/abel/more/index.html
[2] subsampling is an unreleased feature of *vsearch*

4.5 Benchmarking with Other Tools – Command Lines and Software Versions

For benchmarking we have selected the following tools for comparison, listed with their version, source and commands.

- swarm
  version: 2.1.1
  source: https://github.com/torognes/swarm
  commands:
  ```
  ./swarm -t 16 BioMarKs.fas -o output -d 1 -a
  ./swarm -t 16 BioMarKs.fas -o output -d 3
  ```

- CD-HIT
  version: 4.6.1
  source: https://code.google.com/p/cdhit/
  commands:
  ```
  ./cd-hit -M 0 -T 0 -i BioMarKs.fas -o output -c 0.97
  ./cd-hit -M 0 -T 0 -i BioMarKs.fas -o output -c 0.95
  ```

- usearch
  version: 8.0
  source: http://www.drive5.com/usearch/download.html
  commands:
  ```
  ./usearch -cluster_fast BioMarKs.fas -threads 16 -id 0.97
  -centroids cent -uc uc
  ./usearch -cluster_fast BioMarKs.fas -threads 16 -id 0.95
  -centroids cent -uc uc
  ```

- vsearch
  version: 1.1.3
  source: https://github.com/torognes/vsearch
  commands:
  ```
  ./vsearch --cluster_fast BioMarKs.fas --threads 16 --id 0.97
  --centroids cent --uc uc
  ./vsearch --cluster_fast BioMarKs.fas --threads 16 --id 0.95
  --centroids cent --uc uc
  ```

- DNACLUST
  version: svn revision 54
  source: http://dnaclust.sourceforge.net/
  commands:
  ```
  ./dnaclust -t 16 --approximate-filter -s 0.97 BioMarKs.fas
  ./dnaclust -t 16 --approximate-filter -s 0.95 BioMarKs.fas
  ```

- sumaclust
  version: 1.0.01
  source: http://metabarcoding.org/sumatra
  commands:

```
./sumaclust -t 0.97 -p 16 BioMarKs.fas -F output
./sumaclust -t 0.95 -p 16 BioMarKs.fas -F output
```

Swarm is a single-linkage hierarchical clustering program while all the other tools are using heuristic and greedy clustering methods. These heuristic and greedy clustering methods use the terms "similarity" to group sequences into cluster, where “each sequence in the cluster must have a similarity above a given identity threshold (radius of a cluster) from the centroid”[3]. For these tools the definition of "cluster" is quite different with single-linkage hierarchical clustering, where the radius of a cluster is not defined and rather depends on the characteristics of the dataset. Therefore, the comparison will be done using two types of settings for each type of clustering tools: threshold 0.97 and 0.95 for the heuristic clustering tools; $d = 1$ and $d = 3$ for the hierarchical clustering tools. However due to the difference in cluster definition, these settings are not directly comparable.

[3] http://drive5.com/usearch/manual/uclust_algo.html

Chapter 5

Results and Discussion

The program in this project is denoted as p-swarm and all tests in this chapter were executed using *5-mers* for k-mer vector comparison. This chapter contains the presentation, evaluation, and discussion of the test result of p-swarm executed on different datasets and parameters, and also comparison of p-swarm with swarm and some of the popular heuristic clustering tools, i.e. CD-HIT, USEARCH, VSEARCH, DNACLUST, and *sumaclust*.

5.1 Comparison of Speedup and Memory Consumption on Brute-Force, First-Level, and Multiple-Level

In the previous chapter about design, we proposed two economic search methods for omitting redundant calculations. Here we will present the results of running three different variants of p-swarm, namely: (i) without omitting calculations – labelled as *brute-force*, (ii) economic search on first level of connection – labelled as *first-level*, and (iii) economic search on multiple level of connection – labelled as *multiple-level*.

To observe the improvement of performance when more threads or processors are added, each variants of p-swarm were executed on BioMarKs dataset using $d = 1$ and $d = 3$ on several numbers of threads $t = 1, 2, 4, 8, 9, 12, 14, 16$. Table 5.1 shows the execution time of each variant while table 5.2 shows the comparison of speedup that was obtained on the each number of threads. Speedup is calculated by $speedup = T_1 / T_n$ where T_1 is the execution time if using a single thread, and T_n is the execution time if using n threads. In this experiment, *multiple-search* were set to perform the economic search up to the sixth level of connections.

As shown on the execution time comparison chart (see figure 5.1) for $d = 1$ the *first-level* method consistently performed better than the *brute-force* method with the improvement of 30% less execution time. When using the economic search on multiple level connections, p-swarm

performs at least twice better than the *first-level* method, and at least three times better than the *brute-force* method. The same result also applies for $d = 3$.

	$d = 1$ time (sec)			$d = 3$ time (sec)		
threads	BF	FL	ML	BF	FL	ML
1	1,051.58	647.19	307.04	3,224.07	521.65	231.80
2	554.04	338.34	160.02	1,628.28	267.11	122.53
4	284.18	173.05	81.51	737.20	135.83	64.53
8	158.16	94.90	43.97	453.55	70.97	34.53
9	144.18	86.07	39.81	413.45	64.52	31.51
12	116.94	69.88	31.86	332.80	51.13	25.91
14	107.16	63.82	28.65	308.70	47.31	24.26
16	98.22	59.09	26.96	300.45	44.01	23.44

Table 5.1: Execution time of *brute-force*, *first-level*, *multiple-level* with various thread numbers

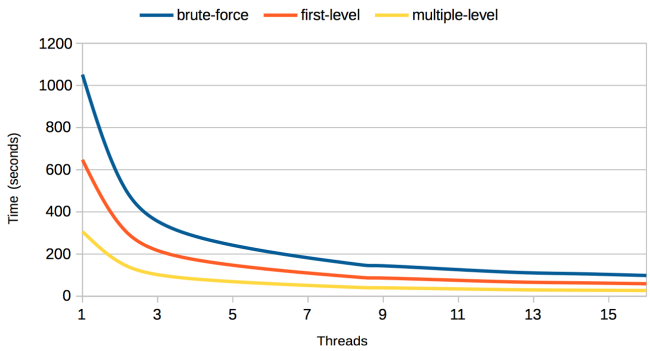


Figure 5.1: Evaluation of execution time on *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 1$

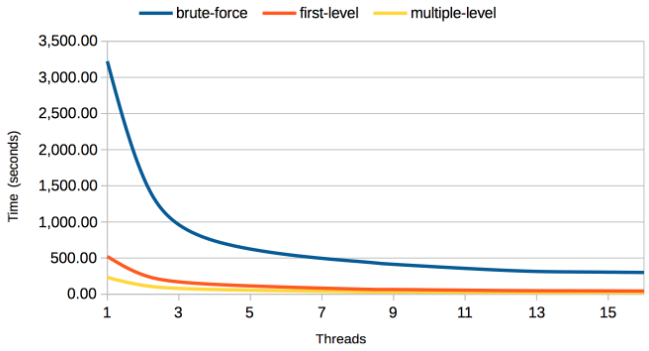


Figure 5.2: Evaluation of execution time on *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 3$

All methods gave consistent speedup when $d = 1$ where the maximum speedup that can be achieved with 16 threads is 11 times in average. By Amdahl's law, this means that in theory the serial fraction of the program stands merely 3% (as shown on equation 5.1). For the value $d = 3$ the speedup of the *multiple-level* method is only 10 times while the speedup for the *first-level* method is almost 12 times. This indicates with a larger value of d one could gain a better speed up if using the $first-level$ method where the parallelism has a finer granularity.

$$\begin{aligned} T_n &= T_1\left(S + \frac{1}{n}(1-S)\right) \\ \frac{1}{11} &= S + \frac{1}{16}(1-s) \\ S &= 0.03 \end{aligned} \tag{5.1}$$

	$d=1$ $speedup = T_1/T_n$			$d=3$ $speedup = T_1/T_n$		
threads	BF	FL	ML	BF	FL	ML
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.90	1.91	1.92	1.98	1.95	1.89
4	3.70	3.74	3.77	4.37	3.84	3.59
8	6.65	6.82	6.98	7.11	7.35	6.71
9	7.29	7.52	7.71	7.80	8.09	7.36
12	8.99	9.26	9.64	9.69	10.20	8.95
14	9.81	10.14	10.72	10.44	11.03	9.55
16	10.71	10.95	11.39	10.73	11.85	9.89

Table 5.2: Speedups of *brute-force*, *first-level*, *multiple-level* with various thread numbers

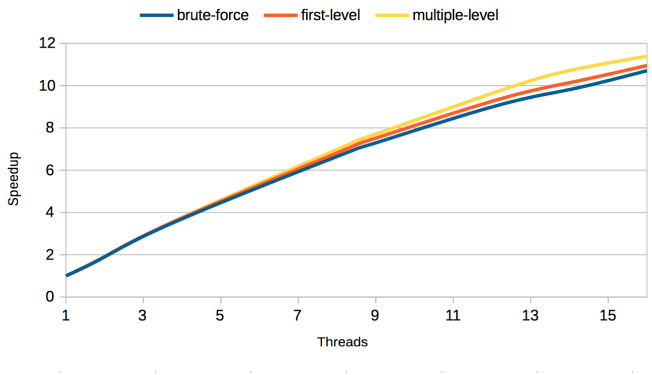


Figure 5.3: Evaluation of speedup of *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 1$

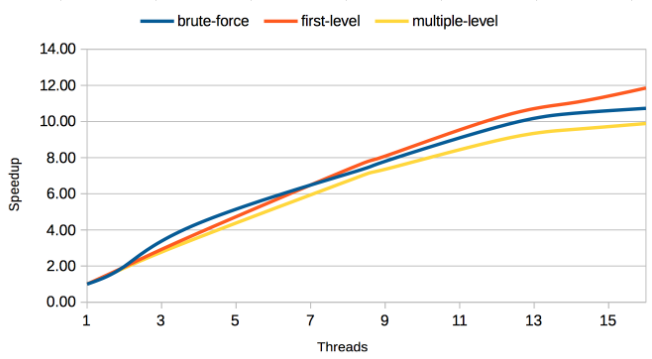


Figure 5.4: Evaluation of speedup of *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 3$

Memory consumption is fairly stable along all methods where adding the number of threads would linearly increase the memory consumption. This is because using more threads requires the allocation of more temporary space to store batches of sequence alignment information. Theoretically both economic search methods — *first-level* and *multiple-level* — should require more temporary space to store next level connections and a set of candidate sequences (as explained in chapter 3), but test result shows only slight difference between the three methods. This is because the number of pairwise alignments that were performed in the brute-force method is more than the ones in the *first-level* and *multiple-level*, so that more memory was used for this occassion.

	$d = 1$ memory (MB)			$d = 3$ memory (MB)		
threads	BF	FL	ML	BF	FL	ML
1	197	196	198	192	193	191
2	203	201	203	197	198	195
4	216	216	217	211	208	211
8	233	237	234	238	229	233
9	248	221	238	245	234	239
12	259	237	253	250	249	259
14	258	236	261	260	252	262
16	274	249	273	275	265	268

Table 5.3: Memory consumptions of *brute-force*, *first-level*, *multiple-level* with various thread numbers

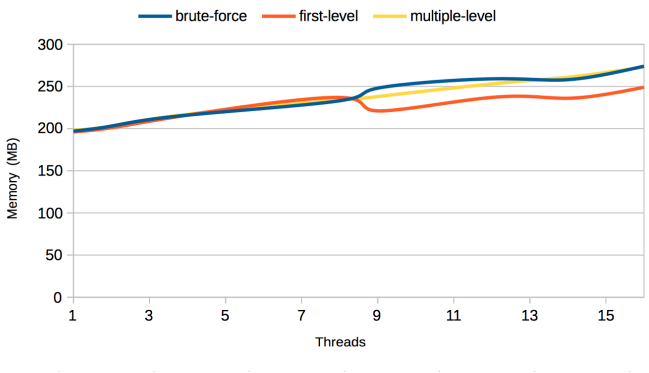


Figure 5.5: Evaluation of memory consumptions of *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 1$

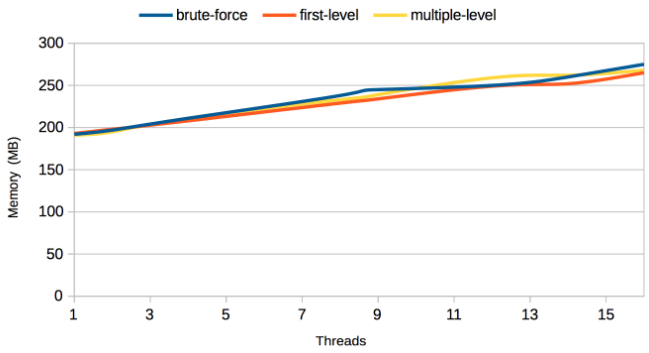


Figure 5.6: Evaluation of memory consumptions of *brute-force*, *first-level*, *multiple-level* with various thread numbers for $d = 3$

5.2 Comparison of Execution Times and Memory Consumptions on Datasets of Different Average Length

Both variants – *first-level* and *multiple-level* – were executed to cluster four datasets each with a different average length that each contains averagely 300.000 unique sequences. More details about the attribute of each dataset are presented on table 5.4. The BioMarKs dataset originally consists of the desired amount of unique sequences, while other datasets were subsampled to obtain a similar amount of unique sequences.

dataset [a]	total nt pre-drp.[b]	total nt post-drp.[c]	unique sequences	average length
EMP_0.0006	8.65×10^7	4.06×10^7	352,211	115
TARA_0.01	5.74×10^9	4.54×10^7	354,020	128
BioMarKs	5.63×10^8	1.19×10^8	312,503	381
greengenes_0.35	4.66×10^8	4.66×10^8	331,831	1,404

[a] number in the file name represents the proportion that was subsampled from the dataset.
[b] total nucleotides before dereplication
[c] total nucleotides after dereplication

Table 5.4: Dataset of different average length

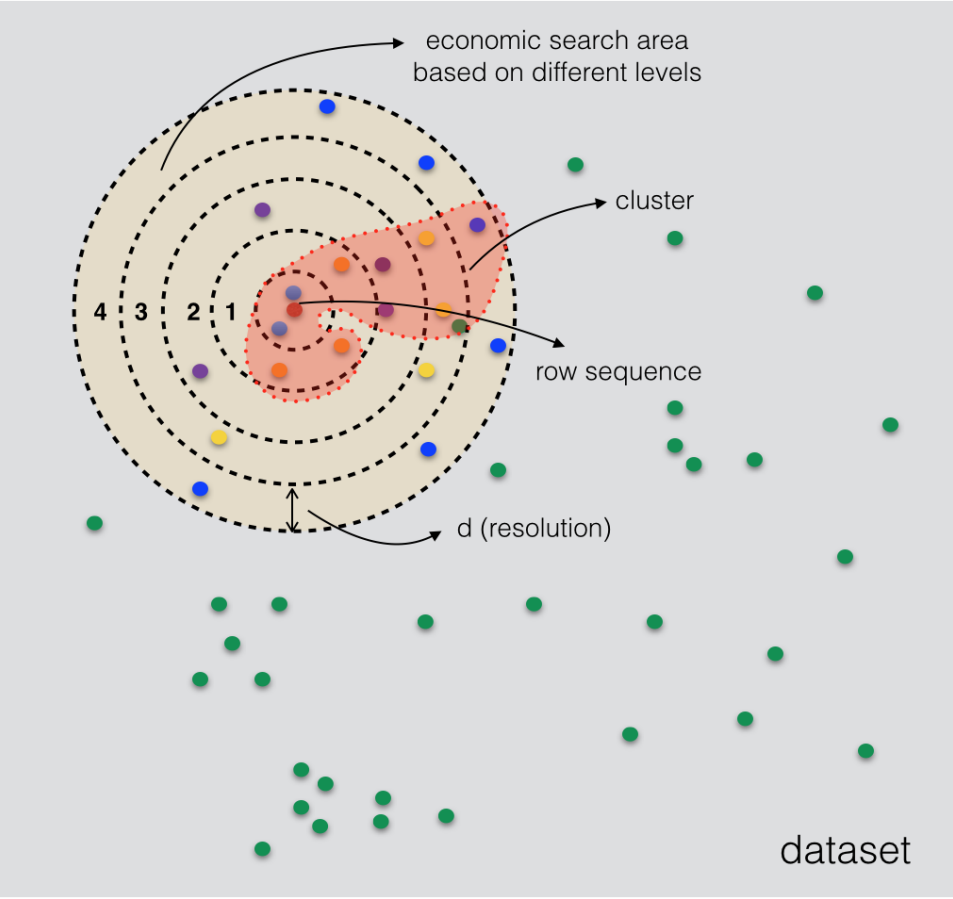
Previous experiments on BioMarKs have shown time improvement of *multiple-level* over *first-level* for $d = 1$ and $d = 3$, which apparently is not the case for datasets with short sequences such as EMP and TARA dataset. Table 5.5 shows that multiple-level starts to perform worse than *first-level* when $d \geq 2$ for EMP dataset, and at $d \geq 3$ for TARA dataset. These results correlate with the fact that EMP dataset is averagely shorter than TARA dataset.

Figure 5.7 shows the illustration of four level economic search where the row sequence is connected to two other sequences. The economic search area is the area that contains all sequences with the edit distance less than $5d$ with the row sequence. Within this search area, sequences might or might not belong to the cluster that is connected with the row sequence. The cluster in this illustration is denoted with the red outline.

When the value of d is higher, the search area is also bigger and therefore it includes more sequences as the candidates. This yields to a higher overhead during the management of economic search flags. When the dataset is comprised of long sequences, it is still more cost-effective to use the *multiple-level* method since the cost to compare the k-mer profiles and to align longer sequences is even greater than the overhead that is caused by the economic search flags management. This is based on the fact that the time for performing pairwise alignment — denoted as $\mathcal{O}(N^2)$ — increases quadratically to the length of the

sequence. However for datasets with short sequences like TARA and EMP, the overhead turns out to be more expensive than the cost of k-mer comparison and sequence alignment. This premise is aligned with the findings in section 5.5, table 5.14 where the overhead is higher for a dataset with shorter sequences.

In other words, the advantage of using the *multiple-level* method becomes counterproductive when the cost to estimate and to calculate the proximity between two sequences is even cheaper than the cost to manage the flags for the economic search. The cost mentioned here is influenced by the size of the search area, where a larger d would lead to a larger search area.



Note: This figure represent an example of one "row calculation" in the *multiple-level* economic search method.

Figure 5.7: Four Level Economic Search

	$d = 1$ time (sec)		$d = 2$ time (sec)		$d = 3$ time (sec)	
average length	FL	ML	FL	ML	FL	ML
115	68.76	59.95	52.56	59.16	46.87	95.22
128	44.38	26.82	24.15	20.67	21.39	30.73
381	59.09	26.96	47.56	21.56	44.01	23.44
1404	98.35	97.70	122.78	116.53	179.09	153.30

Table 5.5: Execution times of dataset of different average length

We observed that the execution time for EMP dataset is longer than the one for TARA dataset although EMP dataset is comprised of shorter sequences than TARA dataset. This is related to the dataset characteristics where a more scattered dataset tends to require more time for clustering. A scattered dataset would contain more sequences that are singletons and do not form any cluster with the other sequences. The economic search is not applicable for these sequences since they do not have any connected members, therefore a full scan must be performed for each of the singletons. This explains why it requires a longer time to cluster these type of datasets. Table 5.6 shows the ratio of total sequences against the total clusters in the dataset, where a lower ratio means that the data is more scattered. Greengenes dataset having the lowest ratio appears to be the most scattered, followed by EMP, BioMarks, and TARA.

	$d = 1$		$d = 2$		$d = 3$	
average length	cluster count	ratio *	cluster count	ratio *	cluster count	ratio *
115	203,148	1.73	144,517	2.44	109,080	3.23
128	84,370	4.20	47,309	7.48	33,571	10.55
381	89,745	3.48	48,498	6.44	34,456	9.07
1,404	305,944	1.08	258,903	1.28	220,944	1.50

* ratio calculated as (unique sequences : cluster count).

Table 5.6: Clustering results of dataset of different average length

We can see from table 5.7 that the memory consumption correlates well with the number of total nucleotides after de-replication. Based on our calculation, the memory consumption is growing linearly to the input size — roughly 3.5 bytes was consumed for every nucleotide of the input dataset.

	$d = 1$ memory (MB)		$d = 2$ memory (MB)		$d = 3$ memory (MB)	
average length	FL	ML	FL	ML	FL	ML
115	149	162	137	191	190	241
128	188	167	160	188	164	266
381	249	273	245	253	265	268
1,404	1,203	1,210	1,204	1,211	1,219	1,206

Table 5.7: Memory consumptions of dataset of different average length

5.3 Comparison of Execution Time and Memory on Datasets of Different Size

We have prepared ten different datasets that were subsampled from each TARA and EMP dataset, ranging from 1.09×10^4 to 9.50×10^6 unique sequences. All datasets were clustered with $d = 1$ and $d = 3$, each using the *first-level* method and the *multiple-level* method. More details of each dataset are listed on table 5.8.

dataset [a]	total nt pre-drp. [b]	total nt post-drp. [c]	unique sequences	average length
TARA_0.0001	5.74×10^6	1.39×10^6	1.09×10^4	128
TARA_0.001	5.74×10^7	8.07×10^6	6.29×10^4	128
TARA_0.01	5.74×10^8	4.55×10^7	3.54×10^5	128
TARA_0.1	5.74×10^9	2.48×10^8	1.93×10^6	128
TARA	5.74×10^{10}	1.20×10^9	9.50×10^6	129
EMP_0.0001	1.44×10^7	8.67×10^6	7.56×10^4	115
EMP_0.0005	7.22×10^7	$3,47 \times 10^7$	3.01×10^5	115
EMP_0.001	1.44×10^8	6.25×10^7	5.42×10^5	115
EMP_0.005	7.22×10^8	2.41×10^8	2.08×10^6	116
EMP_0.01	1.44×10^9	4.29×10^8	3.69×10^6	116

[a] number in the file name represents the proportion that was subsampled from the dataset
[b] total nucleotides before dereplication
[c] total nucleotides after dereplication

Table 5.8: Datasets of different size

	$d = 1$ time (sec)		$d = 3$ time (sec)	
total seqs	FL	ML	FL	ML
TARA_0.0001	0.16	0.18	0.13	0.15
TARA_0.001	1.22	1.17	0.95	1.25
TARA_0.01	44.38	30.73	21.39	26.82
TARA_0.1	1,456.20	661.86	660.88	1,353.00
TARA	35,025.41	13,885.99	17,979.63	56,096.28
EMP_0.0001	3.03	2.64	2.06	3.12
EMP_0.0005	49.12	44.52	33.83	63.38
EMP_0.001	162.28	142.26	111.54	254.09
EMP_0.005	2,238.11	1,175.14	1,541.11	5,813.64
EMP_0.01	6,997.51	5,334.96	4,793.49	20,162.84

Table 5.9: Execution times on datasets of different size

We can see that in this experiment the *first-level* method also outperforms the *multiple-level* method for $d = 3$ which is the same as what has happened in section 5.2 since TARA and EMP dataset are both comprised of short sequences. The run-time of the program increases quadratically along the growth of the problem size. This is expected as explained in section 3.2 that the number of operations that is needed for clustering is generally $\frac{1}{2}(n^2 - n)$. As shown on figure 5.8 and 5.9 the growth of the run-time for both methods and both value of d is almost adjacent with the theoretical $\mathcal{O}(n^2)$ line.

On the case of TARA dataset, the empirical run-time falls somewhat below the theoretical line while for EMP dataset the empirical runtime is very close to the theoretical $\mathcal{O}(n^2)$ line. This result correlates well with the diversity of each dataset where a more diverse dataset would need more time to be clustered. As displayed on table 5.10, the subsamples of EMP dataset are more scattered than TARA dataset and its subsamples.

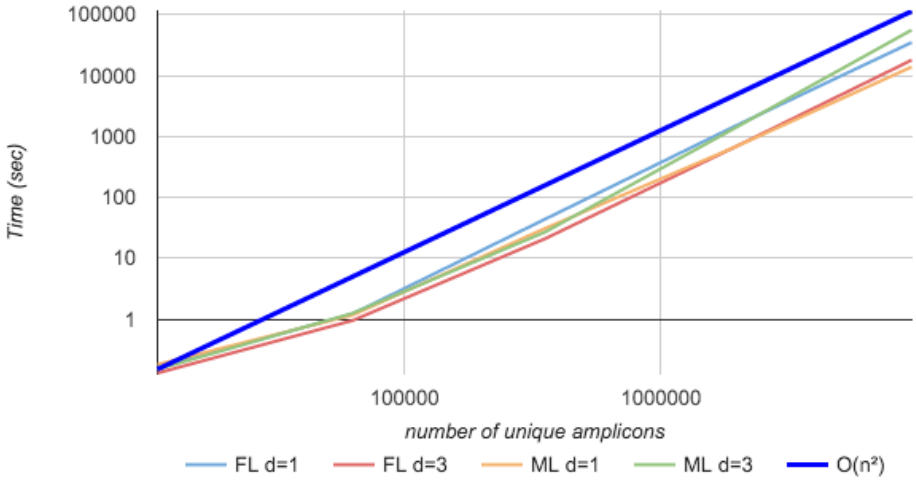


Figure 5.8: Empirical Run-time Growth for TARA dataset

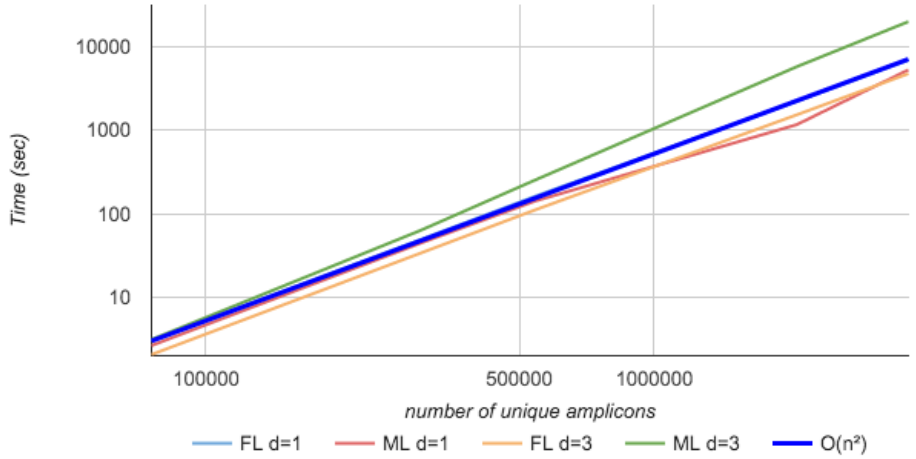


Figure 5.9: Empirical Run-time Growth for EMP dataset

	$d = 1$		$d = 3$	
dataset	cluster count	ratio *	cluster count	ratio *
TARA_0.0001	5,447	2.00	3,356	3.25
TARA_0.001	20,738	3.03	10,213	6.16
TARA_0.01	84,370	4.20	33,571	10.54
TARA_0.1	378,602	5.10	129,988	14.85
TARA	1,646,172	5.77	487,584	19.48
EMP_0.0001	52,148	1.45	33,064	2.29
EMP_0.0005	176,575	1.70	96,467	3.12
EMP_0.001	296,310	1.83	151,470	3.58
EMP_0.005	967,130	2.15	426,760	4.87
EMP_0.01	1,605,171	2.30	665,139	5.55

* ratio calculated as (unique sequences : cluster count).

Table 5.10: Clustering result of different problem size

5.4 Comparison of Execution Time and Memory Consumption on Different Values of d

Table 5.11 is a record of the time, memory and total clusters of BioMarKs dataset when clustered with $d = 1, 2, ..., 10$ using both *first-level* and *multiple-level* methods. We can see from figure 5.10 that the *multiple-level* method always performs better than the *first-level* method for all values of d. The growth of the execution time is almost linear for $1 \leq d \leq 9$. As shown on figure 5.11, the memory consumption for both methods is growing slowly as the value of d increases. In average, the *multiple-level* method consumed slightly more memory than the *first-level* method.

	first-level		*multiple-level*		
d	time (sec)	memory (MB)	time (sec)	memory (MB)	total clusters
1	61.19	269	26.45	244	89,745
2	47.05	269	22.18	249	48,498
3	42.90	273	23.21	257	34,456
4	46.82	279	29.38	288	27,910
5	65.50	288	39.00	326	24,359
6	95.09	288	51.22	349	21,942
7	135.50	290	71.77	343	20,048
8	190.17	294	100.22	356	18,638
9	288.45	304	124.57	370	17,930
10	522.81	313	240.39	387	17,827

Table 5.11: Execution times and memory consumptions for different values of d on BioMarKs dataset

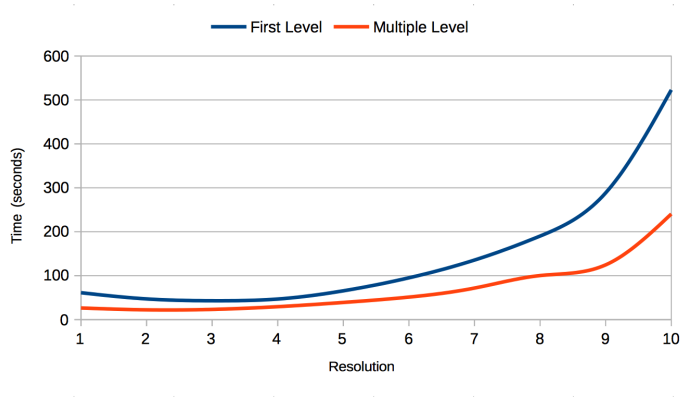


Figure 5.10: Evaluation of execution time for different values of d on BioMarKs dataset

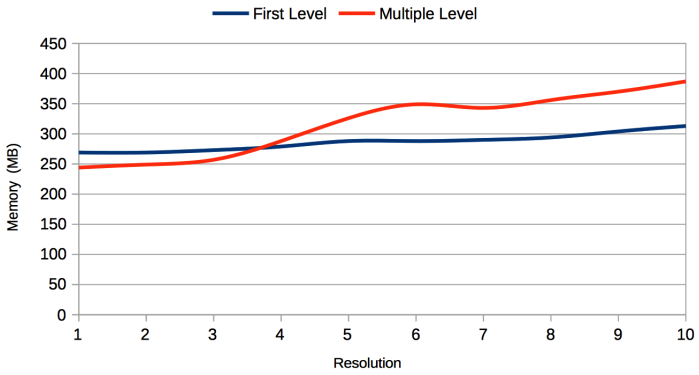


Figure 5.11: Evaluation of memory consumption for different values of d on BioMarKs dataset

The same set of experiment was performed on EMP_0.0005 dataset, as recorded on table 5.12. The difference of characteristics between BioMarKs and EMP_0.0005 datasets can be seen on figure 5.12 where the clustering on BioMarKs dataset produced less number of clusters comparing to ones on EMP_0.0005 dataset although both datasets comprised of the same number of unique sequences (BioMarKs consists of 312,503 sequences and EMP_0.0005 consists of 301,302 sequences).

The *multiple-level* method not only uses more memory for $2 \leq d \leq 8$, but it also took longer time than the *first-level* method (see figure 5.13 and figure 5.14).

	first-level		*multiple-level*		
d	time (sec)	memory (MB)	time (sec)	memory (MB)	total clusters
1	48.97	124	43.47	124	176,575
2	38.44	121	43.37	173	126,605
3	33.85	166	64.06	234	96,467
4	32.81	160	97.55	270	74,543
5	45.38	174	124.36	275	58,342
6	86.67	222	160.71	288	46,208
7	139.29	250	195.52	288	36,962
8	214.61	297	233.34	294	29,936
9	337.2	318	316.07	324	24,477
10	621.56	311	533.4	314	22,120

Table 5.12: Execution times and memory consumptions of different values of d on EMP_0.0005 dataset

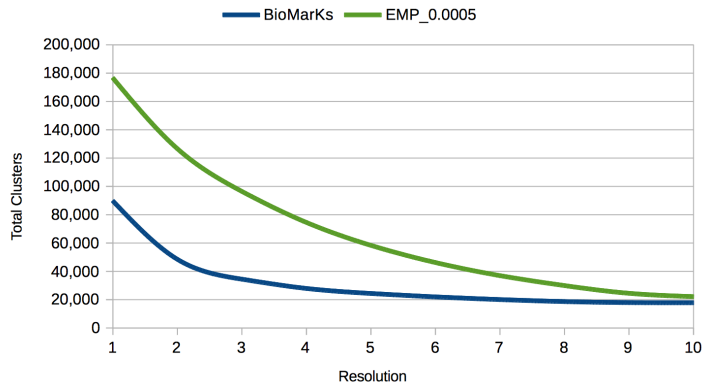


Figure 5.12: Evaluation of total clusters on BioMarKs and EMP_0.0005 dataset for different values of d

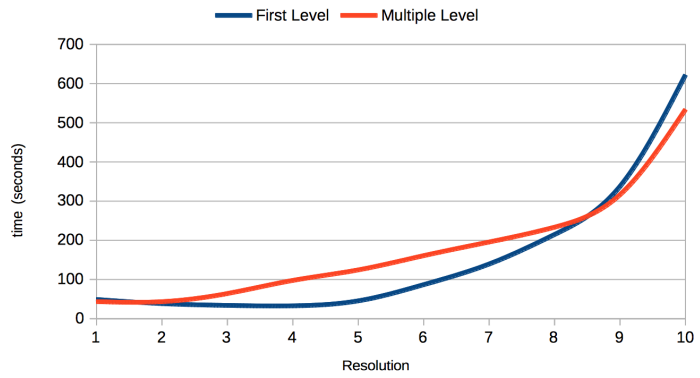


Figure 5.13: Execution time comparison for different values of d on EMP_0.0005 dataset

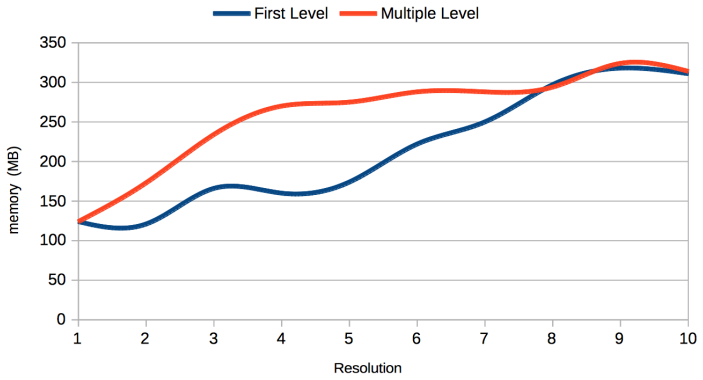


Figure 5.14: Memory comparison for different values of d on EMP_0.0005 dataset

5.5 Analysis of CPU Usage and Time profile Samples

The CPU usage and time profile samples in this section were using Instruments from Apple Xcode version 6.2 on a Macbook Pro with Processor 2,6 GHz Intel Core i7, Memory 16 GB 1600 MHz DDR3.

The CPU usage graph for clustering BioMarKs dataset is presented in figure 5.15. During the initial step — between 00:00 to 00:01 — there was only a single thread that was active. After the initial step, all threads appeared to be busy with a bit of distortion which represent the idle time on some of the threads. Both graphs ended steeply which indicates that all threads finished their job at approximately the same time.

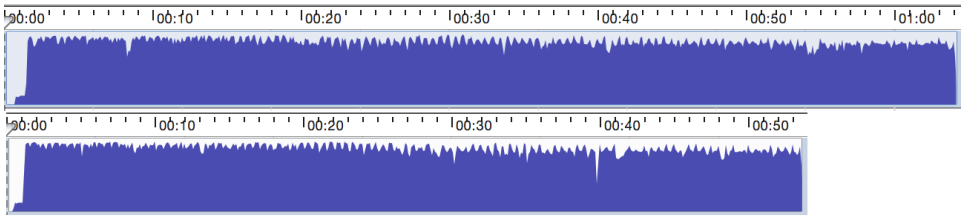


Figure 5.15: CPU usage graph of eight threads on clustering with $d = 1$ and $d = 3$

We have recorded the time profiling informations for $d = 1$ (see table 5.13) and $d = 3$ (see table 5.14) using the *multiple-level* method on BioMarKs dataset and EMP_0.0005 dataset. The time samples for light functions and system libraries were grouped together to summarise a simpler presentation. The records were listed in a descending order from the heaviest task to the lightest task.

Proportion	Time (ms)	Task description
BioMarks (443,084 ms)		
62.70%	277,814	comparing k-mer vector
24.80%	109,993	sequence alignment
8.30%	39,923	light functions and system libraries
2.50%	11,074	fetching sequence by ID
1.60%	7,147	backtrack in sequence alignment
0.10%	481	read sequences from file
EMP_0.0005 (731,727 ms)		
85.60%	626,359	comparing k-mer vector
10.38%	76,105	light functions and system libraries
3.30%	24,253	fetching sequence by ID
0.70%	5,609	sequence alignment
0.01%	530	backtrack in sequence alignment
0.01%	333	read sequences from file

Table 5.13: Time profile summary for $d = 1$

Proportion	Time (ms)	Task description
BioMarks (374,871 ms)		
43.50%	163,069	sequence alignment
43.20%	162,128	comparing k-mer vector
8.40%	35,448	light functions and system libraries
3.40%	13,272	backtrack in sequence alignment
1.40%	5,551	fetching sequence by ID
0.10%	422	read sequences from file
EMP_0.0005 (897,552 ms)		
60.40%	542,122	comparing k-mer vector
34.69%	311,892	light functions and system libraries
2.80%	25,372	sequence alignment
1.30%	12,198	fetching sequence by ID
0.80%	7,193	backtrack in sequence alignment
0.01%	308	read sequences from file

Table 5.14: Time profile summary for $d = 3$

We can see from both tables that the k-mer comparisons and the pairwise alignments on BioMarKs dataset appear to be the heaviest tasks in the clustering process while other administrative tasks occupy only a small portion of the entire run-time. The administrative tasks comprised of the following functions:

- reading sequences from file
- generating the k-mer profile of all sequences
- lookup and assigning sequence into existing cluster, merging clusters, or creating new ones
- fetching and setting the row visit flag
- other functions from system libraries such as: creating and destroying objects, adding elements to `std::vector`, and so on.

For BioMarKs dataset, the proportion of total time consumption of k-mer comparisons and the pairwise alignments are not the same for $d = 1$ and $d = 3$. For $d = 1$ more time was spent on k-mer comparison while for $d = 3$ more time was spent on sequence alignment.

This is not the case for EMP dataset where most time was spent on k-mer comparisons and administrative tasks. Sequence alignment contributes to merely 3.30% and 2.80% of the total runtime. This indicates that using the *multiple-level* search for datasets with short sequences may lead into a higher overhead that is counterproductive to the performance.

To understand more about these numbers, we have also recorded the quantity of k-mer comparisons and pairwise alignments that were

carried out during the clustering operation on the *first-level* method and the *multiple-level* method (see table 5.15 and table 5.16).

	K-mer comparison		Pairwise alignment	
d	Total amount	Percentage*	Total amount	Per 10^4**
		BioMarks		
$d=1$	2.72×10^{10}	55.73	3.41×10^{6}	1.25
$d=3$	8.56×10^{9}	17.54	9.69×10^{6}	11.32
$d=5$	5.39×10^{9}	11.04	1.95×10^{7}	36.20
		EMP_0.0005		
$d=1$	3.24×10^{10}	71.39	3.07×10^{5}	0.09
$d=3$	2.05×10^{10}	45.22	4.90×10^{6}	2.39
$d=5$	1.46×10^{10}	29.94	3.19×10^{7}	21.83

* Ratio against total edges of the dendrogram in percent.
** Ratio against total amount of k-mer comparisons in per ten thousand.

Table 5.15: Operations count of *first-level* method on BioMarKs and EMP_0.0005 dataset

	K-mer comparison		Pairwise alignment	
d	Total amount	Percentage*	Total amount	Per 10^4 *
		BioMarks		
$d=1$	12.70×10^{9}	26.02	1.17×10^{6}	0.92
$d=3$	5.52×10^{9}	11.30	1.93×10^{6}	3.51
$d=5$	4.14×10^{9}	8.48	5.20×10^{6}	12.55
		EMP_0.0005		
$d=1$	2.86×10^{10}	62.92	2.97×10^{5}	0.10
$d=3$	1.73×10^{10}	38.05	4.04×10^{6}	2.34
$d=5$	1.14×10^{10}	23.32	2.20×10^{7}	19.31

* Ratio against total edges of the dendrogram in percent.
** Ratio against total amount of k-mer comparisons in per ten thousand.

Table 5.16: Operations count of *multiple-level* method on BioMarKs and EMP_0.0005 dataset

BioMarKs dataset comprised of $312,503$ unique sequences which means 48.83×10^{9} of pairwise distance computations are needed if clustering is to be performed with the *brute-force* method. The *first-level* economic search was able to omit more than 40% of these computations while the *multiple-level* method was able to omit more than 70% of the distance computations.

A higher value of d leads to less pairwise distance computations but it also yields more false positives in k-mer comparisons. More false-positives lead to more pairs of sequences that need to be verified by the expensive pairwise alignment. From table 5.15 and 5.16 it follows that k-mer comparison filtering method appears to be less effective for a higher value of d and for the dataset with longer sequences.

Increasing k will indeed lead to the exponential growth of complexity since the number of operations that needs to be performed is influenced by the length of the k-mer profile vector, which is 4^k for DNA sequence string. However, a higher k value would produce more precision because longer k-mers are less likely to match by coincidence. For instance in a DNA sequence there are $4^3 = 64$ possible 3-mers, 256 possible 4-mers, 1024 possible 5-mers, and so on. The increase of d might lead to less precision in estimating the edit distance because it increases the value of the allowed different bits, i.e. for 5-mer and $d = 1$, it allows $2dk = 10$ differences, and $d = 3$ allows 30 differences.

5.6 Comparison with Swarm and Other Heuristic Tools

5.6.1 Selection of Tools and Dataset

The program in this thesis is denoted as p-swarm and categorised as a hierarchical method tool, same as swarm. For the heuristic tools, we have selected CD-HIT, USEARCH, VSEARCH, DNACLUST, and sumaclust. Each of the programs was executed to cluster five different datasets ordered by the increasing number of unique sequences: BioMarKs, greengenes, TARA_0.1, EMP_0.01, and TARA. Two types of settings were used for these programs: $d = 1$ and $d = 3$ for the hierarchical tools; $id = 0.97$ and $id = 0.95$ for the heuristic tools. P-swarm was running the *multiple-level* method up to the sixth level to cluster all of the datasets, except in the case of $d = 3$ on EMP and TARA dataset it was running the *first-level* method. All of the tools support multi-threading and were using 16 threads in this experiment. More details about the version and the command line syntax when running each tool can be seen in section 4.5.

As mentioned in section 2.4.3, swarm is an exact, unsupervised, and single-linkage clustering method. Swarm and p-swarm produce exactly the same result set since they have the same concept and algorithm, and differ only on the parallelisation technique. Parallelisation in swarm involves simultaneous k-mer comparisons and pairwise alignments on multiple sequences while the workflow of the clustering is executed chronologically [Mahé et al., 2014].

5.6.2 Results of Benchmarking with Other Programs

The time, memory and clusters count were recorded for each execution as shown on table 5.17, 5.18 and 5.19. The 32-bit version of USEARCH which is free, can only handle up to 4GB of memory, therefore the record for TARA dataset is not available for USEARCH since the memory requirement to cluster this dataset is beyond the limit of the 32-bit version. While for swarm, to cluster TARA dataset with $d = 3$ may take a very long time (more than 3 days), hence the record is not presented here. The same applies for sumaclust for the execution record of greengenes dataset and TARA dataset.

The special algorithm from Swarm for clustering with $d = 1$ has the linear complexity where it took the least time to cluster all of the datasets, especially for the very large datasets the time difference with all other tools becomes very significant. While for $d = 3$ p-swarm performed averagely 10 times better than swarm. Swarm and p-swarm produce the same amount of clusters since they are both single-linkage hierarchical.

We can see from table 5.17 that DNACLUST and sumaclust on average spent the longest time to cluster each of the datasets. CD-HIT and VSEARCH both have nearly the same performance, while USEARCH seems to perform better on greengenes dataset, which is comprised of long sequences. Memory consumption was fairly comparable among all tools, except for DNACLUST and sumaclust which consumed more than twice of the memory comparing to other heuristic tools on most of the datasets.

Memory consumption of p-swarm is either lower than – or comparable with all of the other tools. Apart from that p-swarm is also able to cluster all of the datasets for both settings $d = 1$ and $d = 3$ within a competitive runtime while preserving the quality of being the exact and unsupervised clustering. For example p-swarm is able to cluster the TARA dataset which comprises nearly 10 million unique sequences in less than four hours for $d = 1$ and less than three hours for $d = 3$, while for DNACLUST it took almost one day for $id = 0.97$ and eight hours for $id = 0.95$. As for swarm, it takes only five minutes to cluster the dataset with $d = 1$, but for $d = 3$ it took more than 3 days (the exact time was not recorded in this experiment).

Program	BioMarKs		greengenes		TARA_0.1		EMP_0.01		TARA	
Hiearchical	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$
p-swarm	30	23	795	798	662	661	5,335	4,793	13,887	9,788
swarm	14	269	215	3,726	63	6,440	123	51,766	308	N/A*
Heuristic	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$
CD-HIT	56	43	1,457	5,030	571	220	2,174	1,015	10,824	2,493
USEARCH	71	39	921	391	4,955	1,564	10,719	6,420	N/A**	N/A**
VSEARCH	46	40	1,898	1,518	572	206	3,262	1,619	19,255	4,680
DNACLUST	154	161	2,784	3,670	4,224	2,163	34,586	19,699	69,599	26,991
sumaclust	51	477	N/A*	N/A*	4,210	1,976	34,469	20,724	N/A*	N/A*

* not available due to very long execution time.
** not available due to memory limitation of the free version of USEARCH.

Table 5.17: Execution Time (Seconds) Comparison with Other Clustering Tools

Program	BioMarKs		greengenes		TARA_0.1		EMP_0.01		TARA	
Hiearchical	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$	$d=1$	$d=3$
p-swarm	273	268	2,294	2,278	752	725	1,477	1,377	3,549	2,655
swarm	193	231	2,084	2,214	486	734	905	1,376	2,510	N/A*
Heuristic	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$	$id=0.97$	$id=0.95$
CD-HIT	761	759	2,353	2,325	1,745	1,735	2,898	2,867	4,783	4,677
USEARCH	224	190	2,389	1,893	775	593	1,390	1,149	N/A**	N/A**
VSEARCH	447	448	4,762	4,766	1,182	1,168	2,131	2,145	5,637	5,657
DNACLUST	1,757	1,753	13,834	13,834	2,910	2,928	5,000	5,004	11,744	11,803
sumaclust	997	998	N/A*	N/A*	5,193	5,193	10,248	10,249	N/A*	N/A*

* not available due to very long execution time.
** not available due to memory limitation of the free version of USEARCH.

Table 5.18: Memory (MB) Comparison with Other Clustering Tools

Program	BioMarKs		greengenes		TARA_0.1		EMP_0.01		TARA	
Hiearchical	$d = 1$	$d = 3$	$d = 1$	$d = 3$	$d = 1$	$d = 3$	$d = 1$	$d = 3$	$d = 1$	$d = 3$
p-swarm	89,745	34,456	769,116	539,282	378,602	129,988	1,605,171	665,139	1,646,172	487,584
swarm	89,745	34,456	769,116	539,282	378,602	129,988	1,605,171	665,139	1,646,172	N/A*
Heuristic	$id = 0.97$	$id = 0.95$	$id = 0.97$	$id = 0.95$	$id = 0.97$	$id = 0.95$	$id = 0.97$	$id = 0.95$	$id = 0.97$	$id = 0.95$
CD-HIT	17,745	12,145	85,345	50,906	246,744	104,489	642,872	374,429	1,066,661	391,164
USEARCH	30,617	14,897	127,037	71,152	382,536	148,679	678,506	349,386	N/A**	N/A**
VSEARCH	23,121	14,652	88,278	53,457	284,742	128,629	662,658	358,875	1,254,012	507,357
DNACLUST	25,605	17,750	119,196	71,091	401,701	160,268	935,519	400,070	1,719,720	587,251
sumaclust	21,208	13,833	N/A*	N/A*	340,686	145,241	1,308,838	735,747	N/A*	N/A*

* not available due to very long execution time.
** not available due to memory limitation of the free version of USEARCH.

Table 5.19: Clusters Count Comparison with Other Clustering Tools

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have presented and implemented a parallelised single-linkage hierarchical clustering program named "p-swarm" with the options to run three different variants: *brute-force*, *first-level* and *multiple-level*. This program is available for free with GNU Affero GPL and can be downloaded from the public code repository – GitHub[1]. P-swarm was developed as the parallelised version of "swarm", a robust and fast single-linkage hierarchical clustering for amplicon-based study. Inheriting the same clustering quality as swarm's, our program runs averagely ten times faster than swarm when tested with 16 threads and $d = 3$ on several selected datasets. This result has opened up the possibility to hierarchically cluster a larger volume of dataset, for instance, it only takes a few hours for p-swarm to cluster TARA dataset which comprises almost 10 million amplicons.

We have tested the program on several datasets with different average lengths, using different values of d and we have also observed the growth of time and memory as the problem size increases. The speedup of each variant of p-swarm was 11 times in average when running with 16 threads, which shows a good scalability.

When p-swarm was tested on different problem sizes, it was observed that the growth of the runtime for *first-level* method and *multiple-level* method each matches the theoretical complexity of the algorithm, which is $\mathbb{O}(n^2)$.

The results show that the memory consumption of our program has always been stable for all variants and all experiment sets where it is linear to the input size. In average p-swarm consumes a compatible amount of memory, if not less than swarm and the other heuristic tools. This result demonstrates an efficient utilisation of memory in our program.

[1] https://github.com/mimitantono/p-swarm

Based on the result of the tests it was found out that the economic search could be less effective for clustering datasets that are more scattered. This is caused by the large number of singleton sequences in this type of datasets that each requires a full scan.

It is recommended to cluster datasets with long sequences using the *multiple-level* method for any value of d, while datasets with short sequences may have better performance with the *multiple-level* only for $d \leq 2$. This is based on the fact that the cost to calculate the distance between two short sequences is cheaper than the cost to manage the economic search on a large area (a higher d leads to a larger search area). For this reason, it is recommended to use the *first-level* method for clustering a short sequences dataset especially when $d > 2$.

Based on the profiling information that was collected on p-swarm, we observed that there is a need to achieve a balance between the amount of false positives that were produced by the k-mer comparison and the cost of performing the k-mer comparison itself in order to maximise the performance of the program. A higher value of k may yield less false positives but more costly k-mer comparisons, meanwhile a higher number of false positives would eventually lead to more numbers of expensive pairwise alignments.

We have also compared the runtime and memory consumption of p-swarm with swarm and five other heuristic tools — CD-HIT, USEARCH (32-bit version) , VSEARCH, DNACLUST, and sumaclust — on five different datasets with two types of settings. It was observed that p-swarm was able to handle all of the datasets for both settings, where sumaclust and swarm (for $d = 3$) were not able to cluster within a reasonable amount of time. TARA dataset is the biggest dataset in this experiment that consists of 10 million amplicons. The runtime of p-swarm was found out to be approximately 10 times faster than swarm for $d = 3$ and comparable with other heuristic clustering tools. This result suggests that p-swarm is able to cluster large datasets in a competitive runtime while preserving the quality of being single-linkage hierarchical.

6.2 Recommendation for Future Work

Despite the successful outcome of the parallelisation approach that has been presented in this thesis, it may still be improved in several ways as following.

6.2.1 Extending the Code to Run on Multiple-Nodes Environment

There are still potentials to further improve the speedup by increasing the number of processors. The parallel computing in this thesis is implemented using a low-level API — POSIX threads or pthreads that does not support multiple-nodes computer cluster. Consequently this

program can utilise only the CPU cores in a single node, where most processor families support up to a limited number of CPU cores, e.g. 18 cores for Xeon E5-4669 v3[2] which was recently released in June 2015.

Based on the reason above, replacing the pthreads implementation with an API that supports multi-platform shared memory such as OpenMP[3] or OpenMPI[4] may significantly improve the scalability of p-swarm as it is easier and more economic to get multiple cores on a multiple-nodes computer cluster. However using these sophisticated frameworks also implicates more challenges in managing the shared and local memory not to mention more efforts in debugging during the implementation. Apart from that one would need to consider that the performance of the program may be limited by the communication speed between the nodes, therefore the design of the workflow should be made as asynchronous as possible.

6.2.2 Extending the Algorithm to a Distributed Memory Model

Alternatively the algorithm that was described and implemented in this project can also be extended to a distributed shared memory model where there is no communication needed between one node and another. The brute-force method for the algorithm does not require a shared memory since each row calculation is, in fact, independent and does not need to be executed sequentially. While the economic search can be implemented in the distributed memory model by dividing the input sequences into a set of partitioned datasets that are each assigned to the distributed parallel processor. Each parallel processor is only allowed to perform economic searches on the sequences within the assigned partition. This is to ensure that every row calculation will be processed only once.

The result of the row calculations on each parallel processors is a list containing pairs of connected sequences as described in section 3.6.2. These lists are to be processed in the final step that could be either serial or parallel to build a final result set, followed by finding the singleton sequences.

A distributed memory model like this is suitable for the implementation with a big data distributed processing frameworks like Apache Hadoop MapReduce[5] or other similar technology. For instance, each row calculation can be implemented as a set of "Map" operations that each produces a list of connected pair of sequences as the output. Then these outputs will be sorted and partitioned (if using multiple Reducers) to be processed by the Reducer operation to build a final result set.

[2] http://ark.intel.com/products/85766

[3] http://openmp.org/wp/

[4] http://www.open-mpi.org/

[5] http://hadoop.apache.org/

6.2.3 Extending the Algorithm for Other Types of Hierarchical Clustering

By using the same method for generating pairwise proximities in parallel, this algorithm may be extended to perform other types of hierarchical clustering, such as complete-linkage and average-linkage. However, they involve more challenges in time and space complexity as opposed to single-linkage clustering.

For complete-linkage and average-linkage, the pairwise distance itself needs to be stored instead of only one bit of data as implemented in this project, where 0 represents an unconnected pair of sequences and 1 represents connected sequences. Other than that, the economic search that was described and utilised in this project would not be applicable for complete-linkage nor average-linkage since they do not have the same agglomerative nearest neighbour property or SANN property.

6.2.4 Storing Pairwise Distances as a Cache

Cache is a technique that is widely used in computing to allow the reusing of previous results so that a future request for the same computation may be eliminated. Caching may be implemented for this algorithm by logging the pairwise distances and both IDs of the pair of the sequences into a file as the clustering progresses. This file may be reused in the future to cluster a dataset that share some common amplicons, or to resume an interrupted clustering. During the initialisation, the program would ideally load the cached information by selecting only the pairwise distances which involve amplicons that overlapped with the input dataset. Pairwise distances that were retrieved from the cache may be used to skip some k-mer comparisons and potentially also the expensive pairwise alignments.

Chapter 7

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the april 18-20, 1967, spring joint computer conference* (pp. 483–485). ACM.

Barney, B. et al. (2010). Introduction to parallel computing. *Lawrence Livermore National Laboratory*, *6*(13), 10.

BioPerl. (2014, September). Retrieved from http://www.bioperl.org/wiki/FASTA_sequence_format

Bonder, M. J., Abeln, S., Zaura, E., & Brandt, B. W. (2012). Comparing clustering and pre-processing in taxonomy analysis. *Bioinformatics*, *28*(22), 2891–2897.

Bragg, L. & Tyson, G. W. (2014). Metagenomics using next-generation sequencing. In *Environmental microbiology* (pp. 183–201). Springer.

Cai, Y. & Sun, Y. (2011). Esprit-tree: hierarchical clustering analysis of millions of 16s rrna pyrosequences in quasilinear computational time. *Nucleic Acids Research*, *39*(14), e95–e95.

Caporaso, J. G., Kuczynski, J., Stombaugh, J., Bittinger, K., Bushman, F. D., Costello, E. K., ... Gordon, J. I., et al. (2010). Qiime allows analysis of high-throughput community sequencing data. *Nature Methods*, *7*(5), 335–336.

Chen, W., Zhang, C. K., Cheng, Y., Zhang, S., & Zhao, H. (2013). A comparison of methods for clustering 16s rrna sequences into otus. *PLoS ONE*, *8*(8), e70837.

Dash, M., Petrutiu, S., & Scheuermann, P. (2004). Efficient parallel hierarchical clustering. In *Euro-par 2004 parallel processing* (pp. 363–371). Springer.

DeSantis, T. Z., Hugenholtz, P., Larsen, N., Rojas, M., Brodie, E. L., Keller, K., ... Andersen, G. L. (2006). Greengenes, a chimera-checked 16s rrna gene database and workbench compatible with arb. *Applied and Environmental Microbiology*, *72*(7), 5069–5072.

Edgar, R. C. (2010). Search and clustering orders of magnitude faster than blast. *Bioinformatics*, *26*(19), 2460–2461.

Edgar, R. C., Haas, B. J., Clemente, J. C., Quince, C., & Knight, R. (2011). Uchime improves sensitivity and speed of chimera detection. *Bioinformatics*, *27*(16), 2194–2200.

Flouri, T., Ijaz, U. Z., Mahé, F., Nichols, B., Quince, C., & Rognes, T. (2015). Vsearch. Retrieved from https://github.com/torognes/vsearch

Fu, L., Niu, B., Zhu, Z., Wu, S., & Li, W. (2012). Cd-hit: accelerated for clustering the next-generation sequencing data. *Bioinformatics*, *28*(23), 3150–3152.

George, I., Stenuit, B., Agathos, S., & Marco, D. (2010). Application of metagenomics to bioremediation. *Metagenomics: Theory, Methods and Applications*, 119–140.

Ghodsi, M., Liu, B., & Pop, M. (2011). Dnaclust: accurate and efficient clustering of phylogenetic marker genes. *BMC Bioinformatics*, *12*(1), 271.

Gilbert, J. A. & Dupont, C. L. (2011). Microbial metagenomics: beyond the genome. *Annual Review of Marine Science*, *3*, 347–371.

Gilbert, J. A., Jansson, J. K., & Knight, R. (2014). The earth microbiome project: successes and aspirations. *BMC Biology*, *12*(1), 69.

Gilbert, J. A., Meyer, F., Antonopoulos, D., Balaji, P., Brown, C. T. [C Titus], Brown, C. T. [Christopher T], ... Field, D., et al. (2010). Meeting report: the terabase metagenomics workshop and the vision of an earth microbiome project. *Standards in Genomic Sciences*, *3*(3), 243.

Glenn, T. C. (2011). Field guide to next-generation dna sequencers. *Molecular Ecology Resources*, *11*(5), 759–769.

Gustafson, J. L., Montry, G. R., & Benner, R. E. (1988). Development of parallel methods for a 1024-processor hypercube. *SIAM journal on Scientific and Statistical Computing*, *9*(4), 609–638.

Huse, S. M., Welch, D. M., Morrison, H. G., & Sogin, M. L. (2010). Ironing out the wrinkles in the rare biosphere through improved otu clustering. *Environmental Microbiology*, *12*(7), 1889–1898.

Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data clustering: a review. *ACM computing surveys (CSUR)*, *31*(3), 264–323.

Karsenti, E., Acinas, S. G., Bork, P., Bowler, C., De Vargas, C., Raes, J., ... Claverie, J.-M., et al. (2011). A holistic approach to marine eco-systems biology. *PLoS Biology*, *9*(10), e1001177.

Kuiper, I., Lagendijk, E. L., Bloemberg, G. V., & Lugtenberg, B. J. (2004). Rhizoremediation: a beneficial plant-microbe interaction. *Molecular Plant-Microbe Interactions*, *17*(1), 6–15.

Larkman, K. (2007, September). Yacht for sale: suited for sailing, surfing, and seaborne metagenomics. GenomeWeb News.

Li, W. & Godzik, A. (2006). Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, *22*(13), 1658–1659.

Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., ... Law, M. (2012). Comparison of next-generation sequencing systems. *BioMed Research International*, *2012*.

Logares, R., Audic, S., Bass, D., Bittner, L., Boutte, C., Christen, R., ... Dunthorn, M., et al. (2014). Patterns of rare and abundant marine microbial eukaryotes. *Current Biology*, *24*(8), 813–821.

MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth berkeley symposium on mathematical statistics and probability* (Vol. 1, *14*, pp. 281–297). California, USA.

Mahé, F., Rognes, T., Quince, C., de Vargas, C., & Dunthorn, M. (2014). Swarm: robust and fast clustering method for amplicon-based studies. *PeerJ*, *2*, e593.

May, A., Abeln, S., Crielaard, W., Heringa, J., & Brandt, B. W. (2014). Unraveling the outcome of 16s rdna-based taxonomy analysis through mock data and simulations. *Bioinformatics*, *30*(11), 1530–1538.

Mercier, C., Boyer, F., Bonin, A., & Coissac, É. (2013). Sumatra and sumaclust: fast and exact comparison and clustering of sequences. *Invited Talks*, 27.

National Human Genome Research Institute. (2015, June). Retrieved from http://www.genome.gov/images/content/cost_megabase_.jpg

Nelson, K. E. (2011). *Metagenomics of the human body*. New York, NY: Springer New York.

Olman, V., Mao, F., Wu, H., & Xu, Y. (2009). Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, *6*(2), 344–352.

Olson, C. F. (1995). Parallel algorithms for hierarchical clustering. *Parallel Computing*, *21*(8), 1313–1325.

Pettersson, E., Lundeberg, J., & Ahmadian, A. (2009). Generations of sequencing technologies. *Genomics*, *93*(2), 105–111.

Polanski, A. (2007). *Bioinformatics*. Berlin New York: Springer.

Rauber, T. & Rünger, G. (2013). *Parallel programming: for multicore and cluster systems*. Springer Science and Business.

Russell, D. J., Way, S. F., Benson, A. K., & Sayood, K. (2010). A grammar-based distance metric enables fast and accurate clustering of large sets of 16s sequences. *BMC Bioinformatics*, *11*(1), 601.

Savage, D. C. (1977). Microbial ecology of the gastrointestinal tract. *Annual Reviews in Microbiology*, *31*(1), 107–133.

Schloss, P. D. & Handelsman, J. (2005). Introducing dotur, a computer program for defining operational taxonomic units and estimating species richness. *Applied and Environmental Microbiology*, *71*(3), 1501–1506.

Schloss, P. D., Westcott, S. L., Ryabin, T., Hall, J. R., Hartmann, M., Hollister, E. B., ... Robinson, C. J., et al. (2009). Introducing mothur: open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and Environmental Microbiology*, *75*(23), 7537–7541.

Sleator, R., Shortall, C., & Hill, C. (2008). Metagenomics. *Letters in Applied Microbiology*, *47*(5), 361–366.

Sun, Y., Cai, Y., Liu, L., Yu, F., Farrell, M. L., McKendree, W., & Farmerie, W. (2009). Esprit: estimating species richness using large collections of 16s rrna pyrosequences. *Nucleic Acids Research*, *37*(10), e76–e76.

Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, *92*(1), 191–211.

Wang, S. & Dutta, H. (2011). Parable: a parallel random-partition based hierarchical clustering algorithm for the mapreduce framework.

Weinstock, G. M. (2011). The human microbiome project. *Handbook of Molecular Microbial Ecology I: Metagenomics and Complementary Approaches*, *1*, 307.

Whitman, W. B., Coleman, D. C., & Wiebe, W. J. (1998). Prokaryotes: the unseen majority. *Proceedings of the National Academy of Sciences*, *95*(12), 6578–6583.

Wikimedia. (2015, January). Retrieved from http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg

Wong, D. W. (2010). Applications of metagenomics for industrial bioproducts. *Metagenomics: Theory, Methods and Applications*, 141–158.

Xu, J. (2006). Invited review: microbial ecology in the age of genomics and metagenomics: concepts, tools, and recent advances. *Molecular Ecology*, *15*(7), 1713–1731.

Chapter 8

Appendix: Code Listing

Source codes that are included in this appendix are the ones that were considered as the essential part of the project. The complete package of source code may be viewed at https://github.com/mimitantono/p-swarm.

This project is using GNU Affero GPL as the license, which means that it is free to copy, distribute, and/or modify provided that the software will be made available with the same freedom to copy, distribute and/or modify.

8.1 clusterdata.h

```
1  /*
2   * clusterdata.h
3   *
4   *  Created on: Mar 1, 2015
5   *      Author: mimitantono
6   */
7
8  #ifndef CLUSTERDATA_H_
9  #define CLUSTERDATA_H_
10
11 #include "scan.h"
12 #include<queue>
13 #include<vector>
14
15 class cluster_data {
16 public:
17         cluster_data();
18         virtual ~cluster_data();
19
20         class scanner scanner;
21
22         int thread_id;
23         unsigned long int matches_found;
24         unsigned long int qgram_performed;
25         unsigned long int scan_performed;
26         unsigned long int row_stat;
27         unsigned long int * iteration_stat;
28
29         std::vector<unsigned long int> targetampliconids;
30         std::queue<unsigned long int> next_step;
31         std::queue<unsigned int> next_step_level;
32         std::vector<unsigned long int> * next_comparison;
33
```

```
34          void write_next_comparison(unsigned long int col, unsigned int
                distance);
35          void reset();
36  };
37
38  #endif /* CLUSTERDATA_H_ */
```

8.2 clusterdata.cc

```
1   /*
2    * clusterdata.cc
3    *
4    *  Created on: Mar 1, 2015
5    *      Author: mimitantono
6    */
7
8   #include "clusterdata.h"
9   #include "property.h"
10  #include <string.h>
11
12  cluster_data::cluster_data() {
13          thread_id = -1;
14          next_comparison = new std::vector<unsigned long int>[Property::
                depth + 1];
15          matches_found = 0;
16          qgram_performed = 0;
17          scan_performed = 0;
18          row_stat = 0;
19          scanner.search_begin();
20          iteration_stat = new unsigned long int[Property::depth + 1];
21          for (unsigned int i = 0; i <= Property::depth; i++) {
22                  iteration_stat[i] = 0;
23          }
24  }
25
26  cluster_data::~cluster_data() {
27          if (next_comparison)
28                  delete[] next_comparison;
29          if (iteration_stat)
30                  delete[] iteration_stat;
31  }
32
33  void cluster_data::reset() {
34          for (unsigned int i = 0; i <= Property::depth; i++) {
35                  std::vector<unsigned long int>().swap(next_comparison[i
                        ]);
36          }
37  }
38
39  void cluster_data::write_next_comparison(unsigned long int col_id,
        unsigned int distance) {
40          if (distance <= Property::max_next)
41                  next_comparison[Property::max_next_map[distance]].
                        push_back(col_id);
42  }
```

8.3 clusterresult.h

```
1   /*
2    * clusterresult.h
3    *
4    *  Created on: Nov 7, 2014
5    *      Author: mimitantono
6    */
7
```

```
8  #ifndef CLUSTERRESULT_H_
9  #define CLUSTERRESULT_H_
10
11 #include<boost/unordered_map.hpp>
12 #include<string>
13 #include<vector>
14
15 typedef struct cluster_info {
16         unsigned long int cluster_id;
17         std::vector<unsigned long int> cluster_members;
18 } cluster_info;
19
20 class cluster_result {
21 public:
22         cluster_result();
23         virtual ~cluster_result();
24         cluster_info * new_cluster(unsigned long int cluster_id);
25         long partition_id;
26         void merge_cluster(cluster_info* cluster, cluster_info* merge);
27         void print(FILE * stream, bool sort);
28         void add_member(cluster_info * cluster, unsigned long int id);
29         cluster_info * find_member(unsigned long int sequence_id);
30 private:
31         boost::unordered_map<unsigned long int, cluster_info> clusters;
32         unsigned long int * member_stat;
33 };
34
35 #endif /* CLUSTERRESULT_H_ */
```

8.4 clusterresult.cc

```
1  /*
2   * clusterresult.cpp
3   *
4   *  Created on: Nov 7, 2014
5   *      Author: mimitantono
6   */
7
8  #include "clusterresult.h"
9  #include "property.h"
10 #include<algorithm>
11 #include "db.h"
12 #include <string.h>
13
14 cluster_result::cluster_result() {
15         partition_id = -1;
16         member_stat = new unsigned long int[Property::db_data.sequences
               ];
17         memset(member_stat, 0, Property::db_data.sequences * sizeof(
               unsigned long int));
18 }
19
20 cluster_result::~cluster_result() {
21         if (member_stat)
22                 delete[] member_stat;
23 }
24
25 cluster_info * cluster_result::new_cluster(unsigned long int cluster_id
       ) {
26         cluster_info info;
27         info.cluster_id = cluster_id;
28         clusters[cluster_id] = info;
29         return &clusters[cluster_id];
30 }
31
32 struct compare_cluster {
```

```
33          inline bool operator()(const std::pair<cluster_info, std::
                vector<unsigned long int> > & struct1,
34                          const std::pair<cluster_info, std::vector<
                                unsigned long int> > & struct2) {
35                  return (Property::db_data.get_seqinfo(struct1.second
                        [0])->header < Property::db_data.get_seqinfo(
                        struct2.second[0])->header);
36          }
37  };
38
39  struct compare_member {
40          inline bool operator()(const unsigned long int id1, unsigned
                long int id2) {
41                  return (Property::db_data.get_seqinfo(id1)->header <
                        Property::db_data.get_seqinfo(id2)->header);
42          }
43  };
44
45  /**
46   * Need to print out consistent format (such as correct result will
         look exactly the same)
47   * this will be an expensive method, turn off except for unit test
48   */
49  void cluster_result::print(FILE * stream, bool sort) {
50          long total = 0;
51          long clust = 0;
52          if (sort) {
53                  fprintf(stderr, "\nResult will be sorted alphabetically
                        \n");
54                  std::vector<std::pair<cluster_info, std::vector<
                        unsigned long int> > > vector_clusters;
55                  for (boost::unordered_map<unsigned long int,
                        cluster_info>::const_iterator cit = clusters.begin
                        (); cit != clusters.end(); ++cit) {
56                          std::pair<cluster_info, std::vector<unsigned
                                long int> > pair;
57                          for (unsigned long i = 0; i < cit->second.
                                cluster_members.size(); i++) {
58                                  pair.second.push_back(cit->second.
                                        cluster_members[i]);
59                          }
60                          pair.first = cit->second;
61                          std::sort(pair.second.begin(), pair.second.end
                                (), compare_member());
62                          vector_clusters.push_back(pair);
63                  }
64                  std::sort(vector_clusters.begin(), vector_clusters.end
                        (), compare_cluster());
65                  for (unsigned int i = 0; i < vector_clusters.size(); i
                        ++) {
66                          for (unsigned int j = 0; j < vector_clusters[i
                                ].second.size(); j++) {
67                                  fprintf(stream, "\n%s", Property::
                                        db_data.get_seqinfo(vector_clusters
                                        [i].second[j])->header);
68                                  total++;
69                          }
70                          fprintf(stream, "\n");
71                          clust++;
72                  }
73          } else {
74                  for (boost::unordered_map<unsigned long int,
                        cluster_info>::const_iterator cit = clusters.begin
                        (); cit != clusters.end(); ++cit) {
75                          for (unsigned long int i = 0; i < cit->second.
                                cluster_members.size(); i++) {
76                                  fprintf(stream, "\n%s", Property::
                                        db_data.get_seqinfo(cit->second.
```

```
                                        cluster_members[i])->header);
77                                  total++;
78                          }
79                          fprintf(stream, "\n");
80                          clust++;
81                  }
82          }
83          fprintf(stream, "\n\nTotal: %ld clusters of %ld sequences",
                clust, total);
84          fprintf(stderr, "Total cluster      : %ld\n", clust);
85          fprintf(stderr, "Total sequence     : %ld\n", total);
86  }
87
88  void cluster_result::merge_cluster(cluster_info* cluster, cluster_info*
        merge) {
89          for (unsigned long int i = 0; i < merge->cluster_members.size()
                ; i++) {
90                  add_member(cluster, merge->cluster_members[i]);
91          }
92          std::vector<unsigned long int>().swap(merge->cluster_members);
93          clusters.erase(clusters.find(merge->cluster_id));
94  }
95
96  cluster_info * cluster_result::find_member(unsigned long int
        sequence_id) {
97          if (member_stat[sequence_id] > 0) {
98                  return &(clusters[member_stat[sequence_id]]);
99          }
100         return NULL;
101 }
102
103 void cluster_result::add_member(cluster_info* cluster, unsigned long
        int sequence_id) {
104         cluster->cluster_members.push_back(sequence_id);
105         member_stat[sequence_id] = cluster->cluster_id;
106 }
```

8.5 cluster.h

```
1  /*
2   * Bigmatrix.h
3   *
4   *  Created on: Dec 24, 2014
5   *      Author: mimitantono
6   */
7
8  #ifndef CLUSTER_H_
9  #define CLUSTER_H_
10
11 #include <pthread.h>
12 #include "clusterresult.h"
13 #include "clusterdata.h"
14
15 class Cluster {
16 public:
17         Cluster();
18         virtual ~Cluster();
19         void find_and_add_singletons();
20         void print_debug(cluster_data ** cluster_data);
21         void print_clusters();
22         void run_thread(cluster_data * cluster_data, int total_thread);
23 private:
24         pthread_mutex_t row_id_mutex;
25         pthread_mutex_t result_mutex;
26
27         cluster_result result;
28         unsigned long int current_row_id;
```

```
29          unsigned long int current_cluster_id;
30
31          unsigned long int get_next_row_id();
32          void add_match_to_cluster(cluster_data * cluster_data, unsigned
                long int row, unsigned long int col);
33          void process_row(bool write_reference, bool use_reference,
                cluster_data * cluster_data, unsigned long int row_id,
34                          unsigned int iteration);
35          void qgram_diff_full_row(unsigned long int row_id, cluster_data
                * cluster_data, bool write_reference);
36          void walkthrough_row_by_reference(unsigned int iteration,
                cluster_data * cluster_data, unsigned long int row_id);
37          void prepare_alignment(unsigned long int col_id, unsigned long
                int row_id, cluster_data * cluster_data);
38  };
39
40  #endif /* CLUSTER_H_ */
```

8.6 cluster.cc

```
1   /*
2    * Bigmatrix.cc
3    *
4    *  Created on: Dec 24, 2014
5    *      Author: mimitantono
6    */
7
8   #include "cluster.h"
9   #include "qgram.h"
10  #include "scan.h"
11  #include "db.h"
12  #include "util.h"
13  #include <locale.h>
14  #include "seqinfo.h"
15  #include "property.h"
16
17  Cluster::Cluster() {
18          current_row_id = 1;
19          current_cluster_id = 1;
20          pthread_mutex_init(&result_mutex, NULL);
21          pthread_mutex_init(&row_id_mutex, NULL);
22  }
23
24  Cluster::~Cluster() {
25          pthread_mutex_destroy(&result_mutex);
26          pthread_mutex_destroy(&row_id_mutex);
27  }
28
29  unsigned long int Cluster::get_next_row_id() {
30          unsigned long int return_row = 0;
31          pthread_mutex_lock(&row_id_mutex);
32          if (current_row_id <= Property::db_data.sequences) {
33                  return_row = current_row_id++;
34          }
35          pthread_mutex_unlock(&row_id_mutex);
36          return return_row;
37  }
38
39  void Cluster::run_thread(cluster_data *data, int total_thread) {
40          if (data->thread_id == 0)
41                  progress_init("Calculating matrix :", Property::db_data
                        .sequences);
42          unsigned long int row_id = get_next_row_id();
43          while (row_id > 0) {
44                  row_id--; //0 means that loop should be finished
45                  if (Property::enable_flag) {
46                          process_row(true, false, data, row_id, 1);
```

```
47                          while (data->next_step.size() > 0) {
48                                  process_row(false, true, data, data->
                                        next_step.front(), data->
                                        next_step_level.front());
49                                  data->next_step.pop();
50                                  data->next_step_level.pop();
51                          }
52                          data->reset();
53                  } else {
54                          process_row(false, false, data, row_id, 1);
55                  }
56                  row_id = get_next_row_id();
57          }
58          if (data->thread_id == 0) {
59                  progress_done();
60          }
61  }
62
63  void Cluster::process_row(bool write_reference, bool use_reference,
        cluster_data * data, unsigned long int row_id, unsigned int
        iteration) {
64          seqinfo_t * row_sequence = Property::db_data.get_seqinfo(row_id
                );
65          if (row_sequence->is_visited()) {
66                  return;
67          }
68          row_sequence->set_visited();
69          ++data->row_stat;
70          ++data->iteration_stat[iteration];
71          if (!use_reference) {
72                  for (unsigned long col_id = row_id + 1; col_id <
                        Property::db_data.sequences; ++col_id) {
73                          seqinfo_t * col_sequence = Property::db_data.
                                get_seqinfo(col_id);
74                          unsigned long qgramdiff = qgram_diff(
                                row_sequence->qgram, col_sequence->qgram);
75                          if (qgramdiff <= Property::resolution) {
76                                  data->targetampliconids.push_back(
                                        col_id);
77                          } else if (write_reference && qgramdiff <=
                                Property::max_next) {
78                                  data->next_comparison[Property::
                                        max_next_map[qgramdiff]].push_back(
                                        col_id);
79                          }
80                  }
81                  data->qgram_performed += Property::db_data.sequences -
                        row_id - 1;
82          } else if (use_reference) {
83                  for (unsigned int j = 0; j <= iteration; ++j) {
84                          std::vector<unsigned long int> new_comparison;
85                          for (unsigned int k = 0; k < data->
                                next_comparison[j].size(); ++k) {
86                                  bool push = true;
87                                  unsigned long int col_id = data->
                                        next_comparison[j][k];
88                                  if (col_id > row_id) {
89                                          seqinfo_t * col_sequence =
                                                Property::db_data.
                                                get_seqinfo(col_id);
90                                          unsigned long qgramdiff =
                                                qgram_diff(row_sequence->
                                                qgram, col_sequence->qgram)
                                                ;
91                                          if (qgramdiff <= Property::
                                                resolution) {
92                                                  data->targetampliconids
                                                        .push_back(col_id);
```

```
93                                                      push = false;
94                                              }
95                                      }
96                                      if (push) {
97                                              new_comparison.push_back(col_id
                                                    );
98                                      }
99                              }
100                             new_comparison.swap(data->next_comparison[j]);
101                             std::vector<unsigned long int>().swap(
                                    new_comparison);
102                     }
103
104             }
105             data->scan_performed += data->targetampliconids.size();
106
107             data->scanner.search_do(row_id, &data->targetampliconids);
108
109             for (unsigned long j = 0; j < data->targetampliconids.size();
                    ++j) {
110                     unsigned long int col_id = data->targetampliconids[j];
111                     unsigned long int diff = data->scanner.master_result[j
                            ];
112                     if (diff <= Property::resolution) {
113                             add_match_to_cluster(data, row_id, col_id);
114                             if (Property::enable_flag && !Property::db_data
                                    .get_seqinfo(col_id)->is_visited() &&
                                    iteration < Property::depth) {
115 //                                  Property::db_data.get_seqinfo(col_id)->
        set_visited();
116                                     data->next_step.push(col_id);
117                                     data->next_step_level.push(iteration +
                                            1);
118                             }
119                     } else if (write_reference || use_reference) {
120                             data->next_comparison[iteration].push_back(
                                    col_id);
121                     }
122             }
123             std::vector<unsigned long int>().swap(data->targetampliconids);
124             if (data->thread_id == 0)
125                     progress_update(current_row_id);
126 }
127
128 void Cluster::find_and_add_singletons() {
129             for (unsigned long int i = 0; i < Property::db_data.sequences;
                    ++i) {
130                     if (result.find_member(i) == NULL) {
131                             cluster_info * added = result.new_cluster(
                                    current_cluster_id++);
132                             result.add_member(added, i);
133                     }
134             }
135 }
136
137 void Cluster::print_clusters() {
138 #ifdef DEBUG
139             result.print(Property::outfile, true);
140 #else
141             result.print(Property::outfile, false);
142 #endif
143 }
144
145 void Cluster::print_debug(cluster_data ** data) {
146             unsigned long int matches_found = 0;
147             unsigned long int qgram_performed = 0;
148             unsigned long int scan_performed = 0;
149             unsigned long int * iteration_stat = new unsigned long int[
```

```
            Property::depth];
150         for (unsigned int j = 0; j < Property::depth; j++) {
151                 iteration_stat[j] = 0;
152         }
153         for (int t = 0; t < Property::threads; t++) {
154                 fprintf(Property::dbdebug, "Row stat [%d]\t\t: %13ld\n"
                        , t, ((*data)[t]).row_stat);
155                 matches_found += (*data)[t].matches_found;
156                 qgram_performed += (*data)[t].qgram_performed;
157                 scan_performed += (*data)[t].scan_performed;
158                 for (unsigned int j = 0; j < Property::depth; j++) {
159                         iteration_stat[j] += (*data)[t].iteration_stat[
                                j + 1];
160                 }
161         }
162         fprintf(Property::dbdebug, "Total match\t\t: %13ld\n",
                matches_found);
163         fprintf(Property::dbdebug, "Total estimate\t\t: %13ld\n",
                qgram_performed);
164         fprintf(Property::dbdebug, "Total search\t\t: %13ld\n",
                scan_performed);
165         fprintf(stderr, "Total estimate     : %ld\n", qgram_performed);
166         fprintf(stderr, "Total search       : %ld\n", scan_performed);
167         for (unsigned int j = 0; j < Property::depth; j++) {
168                 fprintf(Property::dbdebug, "Iteration[%d]\t: %13ld\n",
                        j + 1, iteration_stat[j]);
169         }
170 }
171
172 void Cluster::add_match_to_cluster(cluster_data * data, unsigned long
        int first, unsigned long int second) {
173         pthread_mutex_lock(&result_mutex);
174         cluster_info * existing_first = result.find_member(first);
175         cluster_info * existing_second = result.find_member(second);
176         if (existing_first != NULL && existing_second == NULL) {
177                 result.add_member(existing_first, second);
178 #ifdef DEBUG
179                 fprintf(Property::dbdebug, "Add %ld to cluster %ld\n",
                        second, existing_first->cluster_id);
180 #endif
181         } else if (existing_first == NULL && existing_second != NULL) {
182                 result.add_member(existing_second, first);
183 #ifdef DEBUG
184                 fprintf(Property::dbdebug, "Add %ld to cluster %ld\n",
                        first, existing_second->cluster_id);
185 #endif
186         } else if (existing_first == NULL && existing_second == NULL) {
187                 cluster_info * added = result.new_cluster(
                        current_cluster_id++);
188                 result.add_member(added, first);
189                 result.add_member(added, second);
190 #ifdef DEBUG
191                 fprintf(Property::dbdebug, "Create cluster %ld for %ld
                        and %ld\n", current_cluster_id, first, second);
192 #endif
193         } else if (existing_first != NULL && existing_second != NULL) {
194                 if (existing_first->cluster_id != existing_second->
                        cluster_id) {
195 #ifdef DEBUG
196                         fprintf(Property::dbdebug, "Merge cluster %ld
                                with %ld\n", existing_first->cluster_id,
                                existing_second->cluster_id);
197 #endif
198                         if (existing_first->cluster_members.size() >
                                existing_second->cluster_members.size()) {
199                                 result.merge_cluster(existing_first,
                                        existing_second);
200                         } else {
```

```
                                    result.merge_cluster(existing_second,
                                        existing_first);
                            }
                        }
                    }
#ifdef DEBUG
                    fprintf(Property::dbdebug, "%ld and %ld are connected\n", first
                        , second);
#endif
                    ++data->matches_found;
                    pthread_mutex_unlock(&result_mutex);
}
```